Destruction Testing: Ultra-Low Delay using Dual Queue Coupled Active Queue Management

Henrik Steen

Thesis submitted for the degree of
Master in Informatics: Programming and Networks
60 credits

Department of Informatics
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Spring 2017

Destruction Testing: Ultra-Low Delay using Dual Queue Coupled Active Queue Management

Henrik Steen

Destruction Testing: Ultra-Low Delay using Dual Queue Coupled Active Queue Management

http://www.duo.uio.no/

Printed: Reprosentralen, University of Oslo

Abstract

Large queues on the Internet are traditionally needed to provide high utilization of network traffic, however it comes at the cost of high delay. DualPI2 is an Active Queue Management (AQM) proposal which attempts to allow scalable TCP traffic to co-exist with traditional TCP, allowing both high utilization and low delay. This thesis presents a test framework that can be used to evaluate AQMs and congestion controls, being capable of simulating a full testbed on a single machine. The test framework is used to evaluate DualPI2 trying to find areas where it might break and need improvements. I find that the test framework is very useful in investigating and analyzing the behaviour of DualPI2, and that there are still some scenarios that should be considered for improvements.

Contents

List of Figures

List of Tables

List of Listings

Glossary

AIMD Additive increase/multiplicative decrease. The classical way for congestion controls to behave.. 8

AQM Active queue management. 1, 2

base RTT The roundtrip without congestion. vii, 10, 11, 15, 26, 57, 58

BDP Bandwidth Delay Product. A network with a high BDP is often called a *long fat network*. 8, 19, 48, 79

Congestion avoidance State when the congestion window reaches ssthresh. Halves cwnd to recover, then rate increases by a fixed amount each RTT interval (AIMD). 7, 8, 15

CUBIC A TCP congestion control. The default in Linux. See section 2.2.3 for details. 8, 10, 39, 48, 66, 67

DCTCP Data Center TCP. A scalable congestion control algorithm. See section 2.4 for details. 10, 20

drop probability Same as loss probability. 21, 65

DualPI2 An AQM, see section 3.7. i, vii, viii, 6, 11, 19–21, 25, 27, 31, 58, 59, 63, 65–67, 71–73, 79–81, 84, 89, 92

ECN Explicit Congestion Notification. A flag in the TCP header indicating there is building congestion. See section 2.3. 9, 10, 20, 39

greedy A TCP utility to generate greedy traffic and attempt to fill queues to ensure there is always data available to be sent. See section 7.1. 5, 47

MSS Maximum segment size. The maximum amount of data that can be transmitted in a single TCP segment. This equals the TCP packet excluding the header. With an MTU of 1 500 bytes, MSS is at most 1460 bytes, but usually a timestamp option is present in the TCP header so the MSS then becomes 1 448 bytes. 57

MTU Maxmimum transmission unit. The maximum size of the network layer when sending data. Using Ethernet as the underlying protocol this is usually limited to 1 500 bytes. 1

PIE Proportional Integral controller Enhanced. An AQM, see section 3.6 for details. vii, viii, 17–19, 27, 63, 65–67, 69, 70

RED Random Early Detection. An AQM, see section 3.5 for details. 10, 16, 17

Reno A TCP congestion control. See section 2.2.2 for details. 8, 10

slow start cwnd doubles each RTT interval. Normally starts at 3 packages. Google experimenting of starting at 10. This is the starting point of a TCP connection, and keeps going till it reaches ssthresh, loss or rwnd. When it reaches ssthresh it enters congestion avoidance. 7

TCP Transmission Control Protocol. 5

Part I

Introduction

Chapter 1

Introduction

1.1 Motivation

The bandwidth capacity on Internet has increased greatly over the years, and bandwidth-intensive services such as Youtube and Netflix have become common to use. Delay is in many situations the critical factor limiting performance. Bandwidth is often easy to increase, while improvements in delay often requires changes to protocols and are more difficult to implement. There are no central control of the Internet, and the way communication between nodes works must in most cases be backward compatible not to exclude existing traffic.

Delay mainly occurs due to increasing buffers and queueing or data that is lost in the route from sender to receiver. This is often caused by the network transmitting more data than its capacity. Transmission Control Protocol (TCP), the most common protocol used on Internet, has mechanisms to control the send rate to accommodate this. The adopted mechanisms today do not have any understanding of the actual queueing in the network and is dependant upon package loss to adjust its sending rate, which itself cause delay due to timeouts and retransmissions.

Recently a proposal to resolve this was outlined, by having the network give feedback to the sender when the queue builds up, without causing loss, while still supporting old clients without causing bias in the traffic. This is resolved by using a dual queue active queue manager with a special scheduling algorithm.

1.2 Main contributions

The contributions of this thesis consists of:

- A TCP utility named greedy that provides insight into sending TCP data and ensures we maximize utilization and fill queues, in order to ensure we can fight against it.
- Improvements into instrumentation code that can be built into AQMs to provide metrics about queueing delay and drop statistics.

- A test framework that can be used to evaluate AQMs and congestion control algorithms under a varietly of parameters, also without requireing a physical testbed.
- Improvements to the DualPI2 reference implementation.
- An evaluation of the DualPI2 AQM.

1.3 Outline

The parts of this thesis consists of:

- Chapter 2 gives an overview of the fundamental technology and protocol details that is required to understand the next parts.
- Chapter 3 introduces the main topic of queueing, giving a brief introduction to the essential parts, the issues it causes, some relevant ways of handling it and introducing the DualPI2 active queue management that we will be evaluating.
- Part II presents the testbed we will use for evaluations, as well as describing my way of using it. The part also presents the test framework developed as part of this thesis, gives a overview of running tests in a virtual environment and presents pitfalls to be aware of during testing.
- Part III presents my evaluations of the DualPI2 AQM.
- Part IV concludes my results and lists future work not covered by this thesis.

Chapter 2

Technology background

2.1 IP

Internet Protocol (IP) is the core protocol for modern networking and which the whole modern internet is built on. Its position in the network stack is the internet layer, above the link layer and below the transport layer.

IP consists of IPv4[20] and IPv6[8] as the two current versions. IP has the responsibility to route packets between networks and IP addresses.

2.2 Transmission Control Protocol

Transmission Control Protocol (TCP)[21], commonly referred to as TCP/IP, is the most widely used protocol on the Internet to allow computers to communicate. TCP provides statefull connections, ensures packets arrive in the correct order, retransmits packet loss and provides congestion and flow control.

TCP maintains a so called congestion window. The congestion window defines how much traffic that can be in-flight and not yet acknowledged. The maintaining of this window provides congestion control. Flow control is the term used when the receiver is limiting the traffic in-flight by announcing a receiving window limit.

2.2.1 Congestion control

Congestion control is a result of the problem observed in 1986 called congestion collapse [16]. Congestion collapse happens when queues fills up and connections are retransmitting data, causing eventually only some data to arrive at the receiver. The original problem for this was resolved in 1986 by introducing Congestion avoidance. [15]

Congestion control works primarily by having a congestion window which controls how many packets are allowed to be in the network. When a connection starts, the window is rapidly increased in a state called slow start, until either a threshold is reached or a congestion signal is detected, normally by a packet being dropped. Outside slow start the phase is called Congestion avoidance. In this phase the connection is probing for more capacity by increasing its congestion window. When a congestion signal is observed the congestion window will reduce, originally by half its size.

A number of congestion control algorithms have been developed, and they all use slighly different ways of controlling the congestion window. However, to co-exist with existing implementations and the original definition, they usually have to respond to congestion in a similar way to avoid stavation of other flows.

2.2.2 Reno

TCP New Reno[13], from now just called Reno, is a loss based congestion control. Reno is considered the reference congestion control algorithm to compare to, to achieve what is called TCP friendliness. However, Reno no longer represents the majority of the congestion control algorithms in use.[29]

Reno's algorithm for controlling the congestion window in Congestion avoidance phase works by increasing the window by one for each RTT, and upon each ongestion signal within a RTT halving the window. This is referred to as additive increase/multiplicative decrease (AIMD).

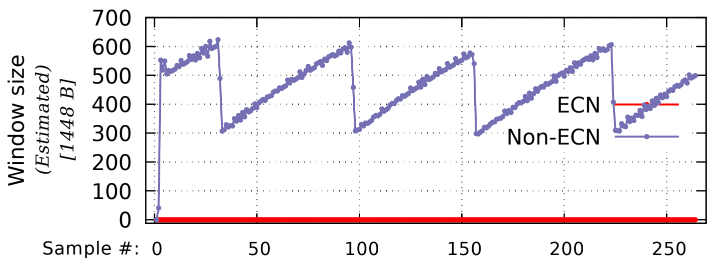


Figure 2.1: Reno's response to congestion signals.

2.2.3 CUBIC

CUBIC[11] is also a loss based congestion control. It is optimized for high speed networks, and its window update algorithm is independent of RTT making it a good fit having a high BDP. CUBIC is the default congestion control algorithm in Linux as of kernel version 2.6.19 released in 2006[17].

CUBIC's window growth function is not linear as with Reno, but uses a cubic function. The result of this is a congestion control quickly increasing its window to a threshold (the window size just before the last congestion signal), staying close to this threshold before quickly probing for more capacity. Upon a congestion signal the window is modified with a factor of 0.7. Having only a small congestion window, CUBIC will fall back to reno-like behaviour to ensure TCP friendliness.

CUBIC also supports ECN instead of drops. Marks with ECN provides the same signal as a drop, except the impairment of drop is avoided.

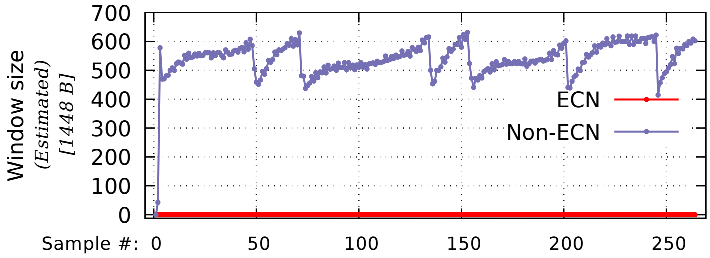


Figure 2.2: CUBIC's response to congestion signals.

2.3 Explicit congestion notification

Explicit congestion notification (ECN) is a feedback mechanism alternative to drops. When ECN is in use and the congested path supports it, packets will be marked with a flag which is returned to the sender in an ACK paket. Classical ECN states that a marked packet should be threated the same as a dropped packet.[23]

When using ECN the congestion control algorithim don't need to wait for a packet drop or selective ACKs to determine a packet was dropped. And most important it don't have to retransmit any packets, which would cause further delay for the connection.

ECN with IPv4

The bits 15 and 16 of the IPv4 header is used for ECN. The codepoints ECT(1) and ECT(0) is currently threated the same. The router can choose to set the CE codepoint instead of dropping if the packet has any of the ECN codepoints.

ECT	CE	RFC 2481 names for the ECN bits
0	0	Not-ECT
0	1	ECT(1)
1	0	ECT(0)
1	1	CE - congestion experienced

Table 2.1: Codepoints for ECN

ECN with IPv6

The last two (least significant) bits of the Traffic Class is used similar as ECN fields.

2.3.1 Using ECN for scalable marking

The downside with current ECN is that a marked packet gives the same response than a dropped packet. This means that the queues still has to build up to the level a packet would be dropped before it will be marked.

Work is going on to change this such that ECN can be used to signal incipient congestion without the congestion control backing off as it would getting a drop.[4] This is called scalable ECN marking.

2.4 Data center TCP

Data center TCP (DCTCP)[2], is a TCP congestion control algorithm which utilizes ECN to provide the extent of queueing rather than only the presence of it as with classical TCP such as Reno and CUBIC, and thus responding more frequently to the congestion signals reducing variance in the sending rate.

DCTCP provides low queueing delay while also giving high utilization, all at the same time without causing impairments such as drops. See figure 2.3 for a visualization. Due to the agressiveness of DCTCP it is currently mainly being used in data centers where the whole network is under control. DCTCP is a lot more agressive than Reno because it expects many congestion signals to reduce the rate as much as Reno. So having DCTCP co-exist with Reno without any other changes would cause Reno traffic to effectively starve. Figure 2.4 shows an example of this.

[7] shows the relation between scalable congestion controls such as DCTCP and classical congestion controls such as Reno and how they can work together by applying different signalling probabilities.

DCTCP requires a change in both the receiver and the router to work properly. The receiver needs to properly echo the CE codepoints so that the sender can receive the proper extent of congestion. The router needs to mark the ECN packets more frequently that it would for a classical TCP connection.

A limitation for using DCTCP outside a data center is its congestion window increase algorithm, which works like Reno increasing by one segment for each RTT. This makes DCTCP less suited having noticable delay caused by base RTT.

DCTCP currently uses the ECT(0) codepoint, while the experimental work on scalable ECN marking is targeting ECT(1). DCTCP today is usually used by configuring the RED AQM to provide proper marking.

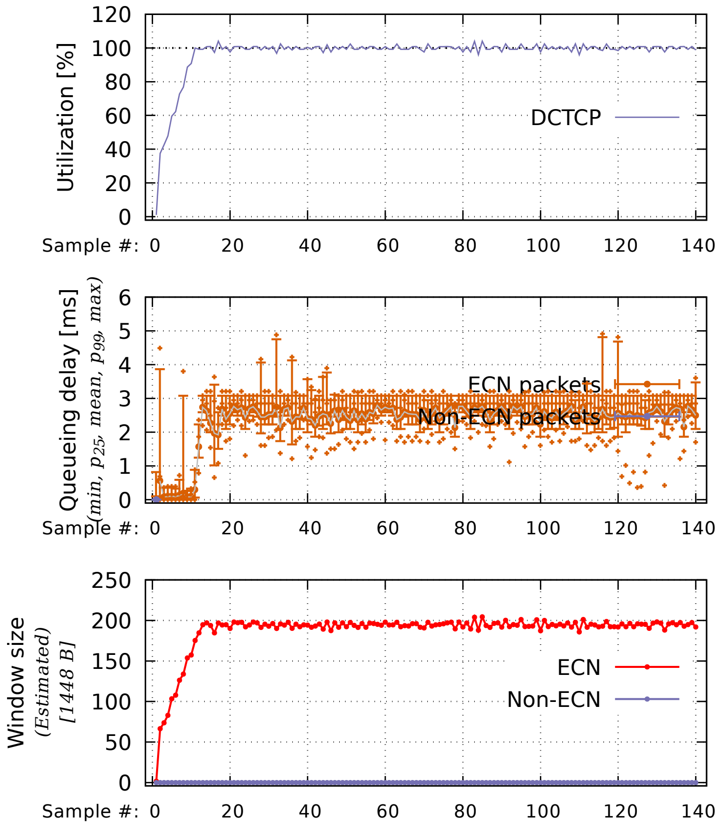


Figure 2.3: DCTCP traffic over 100 Mbit/s with 20 ms base RTT with a target queue delay of 3 ms. Showing full utilization while keeping low delay. DualPI2 used as AQM.

2.5 User Datagram Protocol

User Datagram Protocol (UDP)[22] is a very simple protocol used as an alternative for TCP. UDP is non-responsive, stateless and give no guarantee on ordered data as with TCP. UDP provides no congestion control. The properties of UDP makes it suitable in situations for real time traffic that can handle loss.

As UDP is non-responsive, it can also easily cause overload of not used correctly. In this thesis UDP is the basis of overload, as we can precisely control the rate it is sending, while for TCP the congestion control algorithm will maintain the rate for us.

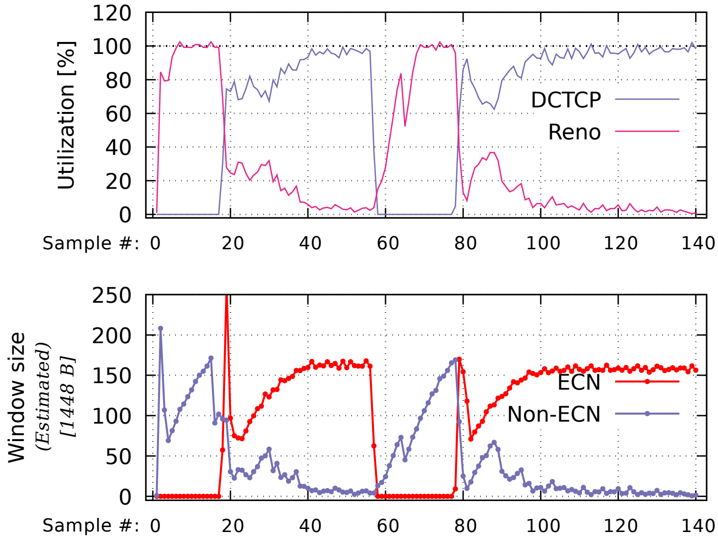


Figure 2.4: DCTCP co-existing with Reno in a single queue with classical ECN marking. The DCTCP flow starts after a short moment and is stopped for a moment durent the test. DCTCP starves the Reno flow.

Chapter 3

Queueing

3.1 Relation between rate and delay

The rate and delay in combination gives the number of bits that can be in flight. Using the Bandwidth Delay Product-formula this can be calculated:

$$\text{bandwidth (b/s)} \times \text{RTT s} = \text{bits in-flight}$$

This formual also gives the window size a TCP connection need to support to be able to utilize the full bandwidth. In Table 3.1 a few example is given to give an understanding of this. Table 3.2 shows what this equals when each packet in the window is of 1 448 bytes.

Achieving 1 Gbit/s while having a RTT of 20 ms will need a window of 2,4 MB. The same value in number of packets of 1 448 bytes is 1 727 packets.

rate \ rtt	1 ms	5 ms	10 ms	20 ms	50 ms	100 ms	200 ms	500 ms
1 Mbit/s	0,1 KB	0,6 KB	1,2 KB	2,4 KB	6,1 KB	12,2 KB	24,4 KB	61,0 KB
50 Mbit/s	6,1 KB	30,5 KB	61,0 KB	122,1 KB	305,2 KB	610,4 KB	1,2 MB	3,0 MB
100 Mbit/s	12,2 KB	61,0 KB	122,1 KB	244,1 KB	610,4 KB	1,2 MB	2,4 MB	6,0 MB
500 Mbit/s	61,0 KB	305,2 KB	610,4 KB	1,2 MB	3,0 MB	6,0 MB	11,9 MB	29,8 MB
1 Gbit/s	122,1 KB	610,4 KB	1,2 MB	2,4 MB	6,0 MB	11,9 MB	23,8 MB	59,6 MB
10 Gbit/s	1,2 MB	6,0 MB	11,9 MB	23,8 MB	59,6 MB	119,2 MB	238,4 MB	596,0 MB

Table 3.1: Window size for various combinations of rates and RTTs in bytes.

rate \ rtt	1 ms	5 ms	10 ms	20 ms	50 ms	100 ms	200 ms	500 ms
1 Mbit/s	0,1	0,4	0,9	1,7	4,3	9	17	43
50 Mbit/s	4,3	22	43	86	216	432	863	2 158
100 Mbit/s	9	43	86	173	432	863	1 727	4 316
500 Mbit/s	43	216	432	863	2 158	4 316	8 633	21 581
1 Gbit/s	86	432	863	1 727	4 316	8 633	17 265	43 163
10 Gbit/s	863	4 316	8 633	17 265	43 163	86 326	172 652	431 630

Table 3.2: Window size for various combinations of rates and RTTs in number of packets of 1 448 bytes.

3.1.1 Common round-trip delay time

Light in vacuum travel at 300,000 kilometers per second, while in fiber this is typically reduced by a factor around 1.44[19], resulting around $207\,756 \frac{km}{s}$. As a rule of thumb the communication will travel at around 200,000 kilometers per second. This equals to one millisecond for every 250 km in fiber. In addition there is processing time throughout the path which adds further time.

According to network details from Verizon[26], traffic between London and New York have an average RTT around 74 ms as of May 2016. According to their statistics the RTT can be as high as above 400 ms (average RTT September 2015 from New Zealand to UK) in their core network. In addition to this there is delay between core network and end points.

3.2 Queueing in routers

Queueing in routes is usually a result of congestion. If the incoming rate is higher than the outgoing rate, there will be queueing. Equation 3.1 shows Little's law which defines average queueing length, L, as the arrival rate, λ, multiplied with the average time each item stay in the queue, W. Queueing causes delay as the data has to use time to sit throught the queue. The amount of queueing depends on the amount of buffer space available, as well as how the queue is managed.

$$L = \lambda W \tag{3.1}$$

Classical TCP (Reno) increase its congestion window by 1 every RTT, and halves the window upon receiving a congestion signal (drop) within a RTT. Without any buffer spacing allowing for queueing within a router, packets will have to be dropped at any congestion. If having perfect pacing of packets this would occur when the link goes above full utilization. Classical TCP would then halve it's congestion window, effectiveley halving the utilization before building up its window again. Without any queueing it would be impossible for classical TCP to utilize the link capacity.

To be able to utilize the link fully, the router need to queue up enough packets so that when a congestion is signalled, the half of the congestion window of the sender still causes full utilization.

Buffer capacity also allows for bursty traffic, without signalling congestion in all cases. This might happen due to wireless links, routing changes, scheduling or other reasons. Scheduling might cause micro bursts that is so short it is not noticable, however the queue will quickly grow and decrease. Without any capacity, packets will be dropped even though there are no real congestion.

Queueing in general might occur other places than in the router itself, such as in the application layer, the TCP implementation in the kernel, ethernet driver and network card, wireless traffic and more. This thesis only focus on the queues caused in the router due to incoming traffic being higher than outgoing link capacity.

3.3 Tail drop

Tail drop is the simplest way of managing a queue. It drops packets trying to enter the queue when it is full, hence the term tail drop. A huge problem with tail drop is that it might cause the queue to remain almost full. Having a big buffer space will allow high utilization, however the delay caused by it is also be very high. Another problem with tail drop is that it might cause synchronization between flows, e.g. multiple flows backing off at the same time, causing under-utilization. [5]

As a simple example of how tail drop works, figure 3.1 shows tail dropping using a small buffer and figure 3.2 shows the same traffic but having a higher buffer. As can be seen from this is the under-utilization with a small buffer and full utilization when the buffer always have data. The queueing delay without much buffer space keeps low, while having a lot of buffer gives a very high queueing delay. The problem with classical TCP is you can't get both high utilization and low queueing delay. In the second example having full utilization, the RTT is varying by a factor of two by the base RTT, from 50 ms to 100 ms only because of queueing.

However, the example shows the best condition for full utilization and queueing. As can be seen from figure 3.2 the queueing delay is close to zero on drops without causing under-utilization. Having a higher buffer size would cause the queueing delay to always be higher than 0 with a long-running flow.

Figure 3.3 shows the same example as figure 3.1, but with CUBIC instead of Reno. Because CUBIC has a more agressive Congestion avoidance algorithm the average utilization is greater than that of Reno.

The examples show only having one single flow at the same time. Having multiple flows will usually improve the utilization as long as the flows don't get a synchronized congestion signal. E.g. if having two flows with similar congestion window and one receiving a congestion signal, the overall window will only reduce by one fourth, not by half.

3.4 Active queue management

Active queue management (AQM) is an advanced form of queue management, an algorithm managing the length of packets queued by marking packages when necessary or appropriate. The algorithm causes congestion signals by the marking which the sender can use to adjust its rate. An AQM also helps ensure there is available buffer capacity for handling burst and avoiding global synchronization. [3] gives recommendations for developing an AQM in today's Internet.

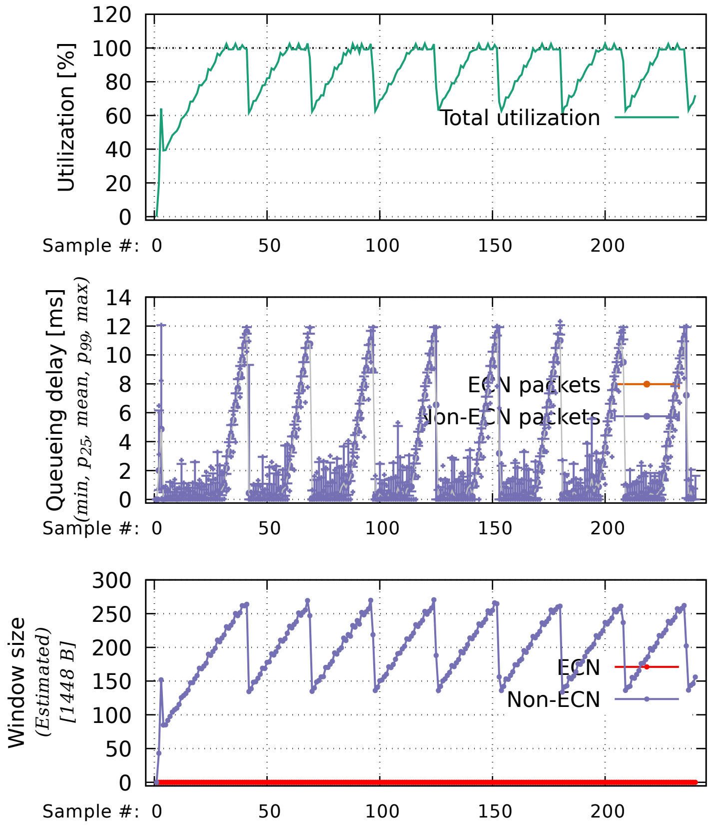


Figure 3.1: Tail dropping with a buffer size of 50 packets. 1 Reno flow. Link rate: 50 Mbit/s. Base RTT: 50 ms.

As apposed to tail dropping, an AQM signals congestion before the queue is full. It also allows for larger buffers for handling bursts, but only using them when needed.

A lot of different AQMs have been developed throughout the years. RED is considered the first AQM, being developed in 1993. An extensive list and insight into different AQMs developed between 1993 and 2011 is given in [1].

3.5 RED - Random Early Detection

In 1993, S. Floyd and V. Jacobsen proposed a mechanism called Random Early Detection (RED) as a possible mechanism for solving the issues caused by tail dropping.[10] RED is an active queue management algorithm which gives feedback to the sender about the network congestion by marking or dropping packages with a probability related to the average queue size.

The RED algorithm is designed where a single marked or dropped package is enough to signal congestion, and as an algorithm that can be deployed gradually. It also ensures a bias against bursty traffic. The

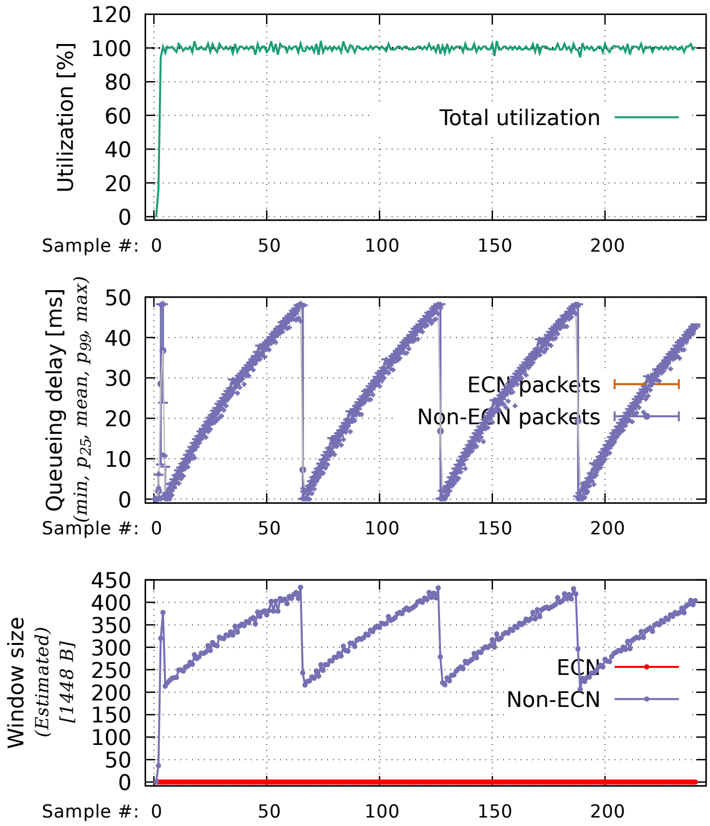


Figure 3.2: Tail dropping with a buffer size of 200 packets. 1 Reno flow. Link rate: 50 Mbit/s. Base RTT: 50 ms.

design also cause the probability of being signalled proportional to that connection's share of the throughput.

The algorithm computes the average queue size. If this is between two thresholds it will calculate a marking probability, linearly between these thresholds related to the average queue size, and increasing the probability more for the count since last marked packet. If this probability occurs the packet will be marked, signalling congestion. If the average queue length is larger than the upper threshold all packages will be marked.

A weakness with RED is that it needs to be properly configured for the case it is deployed. Different link rates and sites will require different configuration. The main problem is that the queue is measured in bytes, not in time.

3.6 PIE - Proportional Integral controller Enhanced

Proportional Integral controller Enhanced (PIE) is an AQM that attempts to keep the queueing delay to a configured value in time. It is

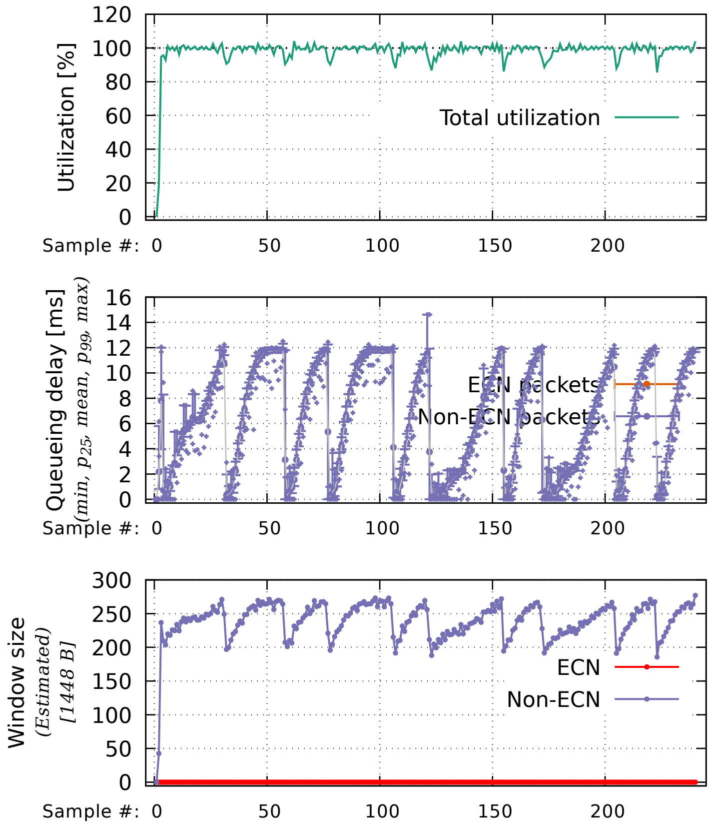


Figure 3.3: Tail dropping with a buffer size of 50 packets. 1 CUBIC flow. Link rate: 50 Mbit/s. Base RTT: 50 ms.

self-tuning and works out of the box in most deployment scenarios.[18] PIE is also the reference AQM for DOCSIS-PIE[28] which is mandatory in DOCSIS 3.1[12], which is the standard used by cable network providers. PIE was made available in the mainline Linux kernel as of January 2014.[1]

PIE uses a Proportional Integral (PI)[14] algorithm as its core to maintain a target queueing delay. It maintains an estimation of dequeue rate and periodically measures the queueing delay from the number of packets in the queue, which is used in the PI controller to calculate a signalling probability. For each packet enqueued the probability is used to determine if a packet should receive a congestion signal.

PIE includes a number of heuristics, e.g. tuning of the probability if it is low to avoid instability, limiting the change in probability and more. Some if these heuristics are discussed in [7].

[1] https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=d4b36210c2e6ecef0ce52fb6c18c51144f5c2d88

3.7 DualPI2

DualPI2, presented in [6], is the AQM being evaluated for this thesis. DualPI2 attempts to solve the problem with queueing delay for all users while keeping the utilization near full. It requires ECN to be able to signal more frequently about congestion without the impairment of drop. To be able to coexist with todays classical TCP it uses a seperate queue for the improved ECN capable traffic. DualPI2 should work under a varietly of conditions, from low-speed networks to high-speed networks with a high delay.

As with PIE, DualPI2 uses the PI controller as its core for controlling the signalling probability. However, while PIE has a tuning table for controlling the PI algorithm, DualPI2 squares the probability before applying it to classical TCP traffic. The analysis in [7] shows that the squaring of the probability achieves the same as the heuristic tune table used in PIE.

Figure 3.4 shows how packets in DualPI2 are processed from enqueue to dequeue. The next sections explains the different parts.

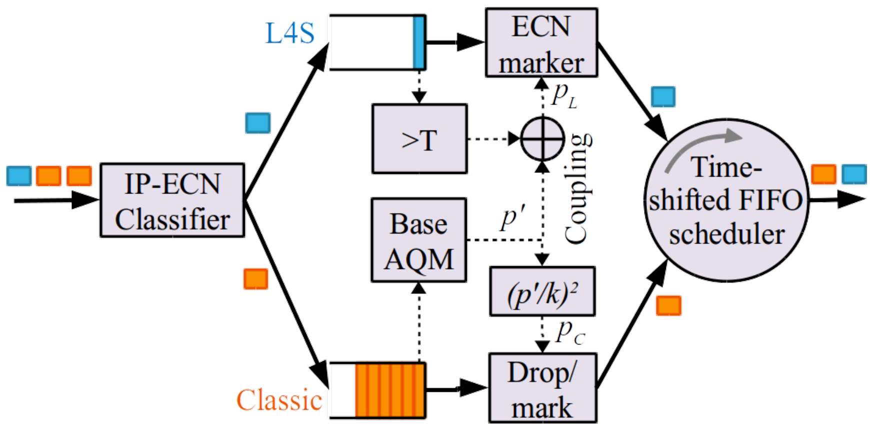


Figure 3.4: Dual Queue Coupled AQM. Figure from the original paper.

3.7.1 Keeping queueing delay low

The rate of congestion signals for classical TCP do not grow as the bandwidth increases. This makes the congestion signalling not scalable, causing a higher variation of the window with a high BDP. Classical TCP signals one congestion signal for each RTT.

To keep queueing delay low, while also having a high utilization, DualPI2 uses a 'scalable' congestion controller in combination of ECN to signal more frequently about congestion. A 'scalable' congestion control algorithm receives a linearly equal amount of congestion signals as the BDP increases. The result is a system that gives a more fine-graded response to congestion, instead of the classical way of responding two one signal for each RTT.

DCTCP is one such algorithm, and is used to test the DualPI2 algorithm, both in the DualPI2 paper and in this thesis.

Using ECN as signalling is essential for DualPI2. ECN effectively gives the same signal as a drop, however by not dropping packets no harm is caused for the flow, such as needing to retransmit data or wait for a possible timeout. Increasing the signalling rate by using drop would cause too much harm to the flows and make them unstable.

To keep queueing delay ultra-low, DualPI2 also uses a low queue threshold for scalable traffic which will mark (never drop) packets that exceed the threshold. The reference implementation uses a value of 1 ms as threshold.

3.7.2 Coupling between classical and scalable TCP

For scalable congestion controls such as DCTCP, the output of the PI algorithm can be used directly for signalling congestion. However, to achieve a balance between unscalable (classical) and scalable TCP, the probability need to be coupled between the two to achieve a fair window balance. The DualPI2 paper shows how this is calculated, and recommends using the coupling factor of 2.

The coupling causes the following probability relationship:

$$p_{classic} = \left(\frac{p_{scalable}}{k}\right)^2$$

With a coupling factor of 2, the probability of 25 % for classical TCP gives the probability of 100 % for scalable TCP. For the reference implementation used in this thesis the probability being calculated is equal to $\frac{p'}{2}$, so it is multiplied by k to to get $p_{scalable}$ and squared to get $p_{classic}$.

3.7.3 Multiple queues

DualPI2 divides traffic into two queues:

Classic queue for packets that do not use scalable congestion controls. I.e. congestion controls suck as Reno and CUBIC, which upon marking/dropping is expected to half the congestion window, or as with CUBIC attempt to provide fairness to Reno halving.

L4S queue, also termed scalable queue, for packets that uses a scalable congestion control, which will measure the amount of signalling feedback and adjust the congestion window by it. Traffic which uses ECT(1), as described in 2.3.1, is used to classify traffic to this queue.

3.7.4 Priority scheduler

DualPI2 uses a time shifted scheduler to allow low queueing in the L4S queue. Without a time shifted scheduler the queueing delay for the two queues would be similar, and it would not allow low latency while still preserving fairness to classic traffic.

The time shifted part of the scheduler work so that if there are packets in both queues, the packet that has spent the longest time is picked, but with an added time shift for L4S traffic:

- The time of the classic packet is kept as is.
- The time of the L4S packet is added the time shift. In the reference implementation of DualPI2 the default time shifted value is 40 ms. This means that a packet in the classic queue has been there 40 ms longer than the packet in the L4S queue the classic packet is dequeued first.

3.7.5 Overload handling

Overload happens due to unresponsive flows causing congestion. The main concern with overload in DualPI2 is the effect it gives for the different queues. Because of the priority scheduler, traffic in the L4S queue will be prioritized as long as the delay in the classic queue is within a specified difference.

Overloading the classic queue will cause the probability to increase causing more drops in the classic queue and more marks in the L4S queue. The L4S queue switches to drop when the marking probability in the L4S queue reaches 100 %, equaling a drop probability of 25 % in the classic queue (having a coupling of 2, see section 3.7.2).

Overloading the L4S queue causes packets in the classic queue to be delayed. Without any traffic in the classic queue, the probability will use the delay in the L4S queue. The increased delay will cause the probability to rise and the overload mechanism will eventually switch from marking to dropping traffic in the L4S queue.

The exact observed behaviour of overloading is part of the main evaluations of this thesis.

Part II

Testbed design and setup

Chapter 4

Topology

To be able to evaluate the DualPI2 AQM I am setting up a simulation network which we can run traffic in and monitor for statistics. Figure 4.1 show how the testbed is structured. The testbed consist of:

- **Two clients:** Each client is connected to a switch and all clients share the same subnet.
- **Two servers:** Each server is connected to a seperate interface on the AQM machine and are in different subnets.
- **AQM:** Machine acting as a router. Has three interfaces. The clients subnet has one interface which is connected to the clients switch. This interface is used to simulate the bottleneck, and is where the scheduler is added.

In additional there is a management network not shown in the figure, where all the machines are directly connected. This is used for control traffic for easier seperation from test traffic.

Testing is done both on a physical testbed as well as in a virtualized environment futher explained in chapter 8. The physical testbed uses 1 GigE network cards and a 1 GigE switch for the clients network. All machines run Ubuntu using 4.10 kernel.

Listing 9 shows the script written used to configure the testbed. Usage of this is shown in the *setup* method in listing 15. The usage of this is further explained in chapter 6 presenting our test framework.

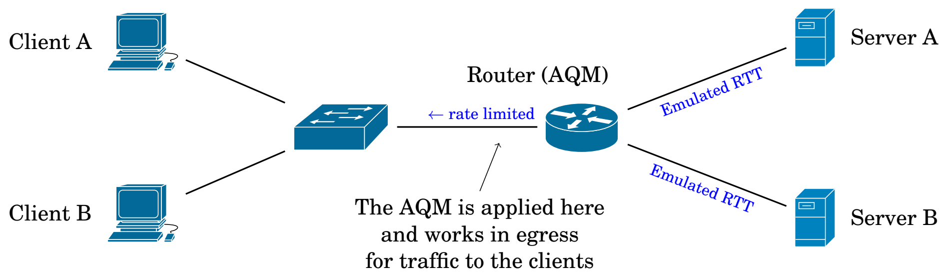


Figure 4.1: The topology used in the testbed.

Simulating a bottleneck

To limit the bandwidth and cause a bottleneck towards the clients, the queueing discipline Hierarchical Token Bucket (HTB) is used. It allows to specify a specific bandwidth and will rate limit the connection.

Simulating base RTT

To simulate base RTT I use *netem*, which will delay packets for a specified time before sending them. Netem is used in each direction on the link to the servers. The servers can be configured with different base RTT independently from each other.

Chapter 5

Collecting metrics

5.1 Measuring actual queueing delay and drops at the AQM

For measuring the actual delay for packets to sit through the AQM as well as dropped packets at the AQM, the packets that leave the AQM have reporting metics added that is then stored when analyzing the packets that leave the interface.

The intial code for this was given me by the people who have performed earlier tests on DualPI2, which I have rewritten and further improved.

To store metrics, the *identification* field of the IPv4 header is replaced. This field consists of 16 bits and is used for segmentation of packets and features not needed. This allows us to inject metrics into the packet and let us analyze this later, without increasing the packet or doing file system operations from the kernel module.

5.1.1 Modifying existing schedulers to add reporting

All the schedulers/AQMs to be tested need to be instrumented to report queueing delay when a packet dequeues as well as incrementing the drop counter which is reported on next dequeued packet.

To make this more easily I have made an API that can be hooked into from the AQMs. Listing 1 shows an example of changes needed to PIE to support the reporting and listing 2 shows the API itself.

```
diff --git a/sch_pie.c b/sch_pie.c
index 5c3a99d..6d57db8 100644
--- a/sch_pie.c
+++ b/sch_pie.c
@@ -30,6 +30,7 @@
 #include <linux/skbuff.h>
 #include <net/pkt_sched.h>
 #include <net/inet_ecn.h>
+#include "testbed.h" /* see README for where this is located */

 #define QUEUE_THRESHOLD 10000
```

```
 #define DQCOUNT_INVALID -1
@@ -74,6 +75,9 @@ struct pie_sched_data {
 	struct pie_vars vars;
 	struct pie_stats stats;
 	struct timer_list adapt_timer;
+#ifdef IS_TESTBED
+	struct testbed_metrics testbed;
+#endif
 };

 static void pie_params_init(struct pie_params *params)
@@ -158,6 +162,12 @@ static int pie_qdisc_enqueue(struct sk_buff *skb, struct Qdisc *sch,

 	/* we can enqueue the packet */
 	if (enqueue) {
+#ifdef IS_TESTBED
+		/* Timestamp the packet so we can calculate the queue length
+		 * when we collect metrics in the dequeue process.
+		 */
+		__net_timestamp(skb);
+#endif
 		q->stats.packets_in++;
 		if (qdisc_qlen(sch) > q->stats.maxq)
 			q->stats.maxq = qdisc_qlen(sch);
@@ -167,6 +177,9 @@ static int pie_qdisc_enqueue(struct sk_buff *skb, struct Qdisc *sch,

 out:
 	q->stats.dropped++;
+#ifdef IS_TESTBED
+	testbed_inc_drop_count(skb, &q->testbed);
+#endif
 	return qdisc_drop(skb, sch, to_free);
 }

@@ -445,6 +458,9 @@ static int pie_init(struct Qdisc *sch, struct nlattr *opt)
 	pie_params_init(&q->params);
 	pie_vars_init(&q->vars);
 	sch->limit = q->params.limit;
+#ifdef IS_TESTBED
+	testbed_metrics_init(&q->testbed);
+#endif

 	setup_timer(&q->adapt_timer, pie_timer, (unsigned long)sch);

@@ -517,6 +533,9 @@ static struct sk_buff *pie_qdisc_dequeue(struct Qdisc *sch)
 		return NULL;

 	pie_process_dequeue(sch, skb);
+#ifdef IS_TESTBED
+	testbed_add_metrics(skb, &((struct pie_sched_data *)qdisc_priv(sch))->testbed);
+#endif
 	return skb;
 }
```

Listing 1: Patch to PIE to add metrics reporting. PIE available in Linux kernel as of version 4.10 is used. Full code available at https://github.com/henrist/aqmt-pie-scheduler.

```
/* This file contains our logic for reporting drops to traffic analyzer
 * and is used by our patched versions of the different schedulers
 * we are using.
 *
 * It is only used for our testbed, and for a final implementation it
 * should not be included.
 */

#include <net/inet_ecn.h>
#include "numbers.h"

/* This constant defines whether to include drop/queue level report and other
 * testbed related stuff we only want while developing our scheduler.
 */
#define IS_TESTBED 1

struct testbed_metrics {
        /* When dropping ect0 and ect1 packets we need to treat them the same as
         * dropping a ce packet. If the scheduler is congested, having a seperate
         * counter for ect0/ect1 would mean we need to have packets not being
         * marked to deliver the metric. This is unlikely to happen, and would
         * cause falsy information showing nothing being dropped.
         */
        u16     drops_ecn;
        u16     drops_nonecn;
};

void testbed_metrics_init(struct testbed_metrics *testbed)
{
        testbed->drops_ecn = 0;
        testbed->drops_nonecn = 0;
}

void testbed_inc_drop_count(struct sk_buff *skb, struct testbed_metrics *testbed)
{
        struct iphdr* iph;
        struct ethhdr* ethh;

        ethh = eth_hdr(skb);

        /* TODO: make IPv6 compatible (but we probably won't going to use it in our testbed?) */
```

```
        if (ntohs(ethh->h_proto) == ETH_P_IP) {
                iph = ip_hdr(skb);

                if ((iph->tos & 3))
                        testbed->drops_ecn++;
                else
                        testbed->drops_nonecn++;
        }
}

u32 testbed_get_drops(struct iphdr *iph, struct testbed_metrics *testbed)
{
        u32 drops;
        u32 drops_remainder;

        if ((iph->tos & 3)) {
                drops = int2fl(testbed->drops_ecn, DROPS_M, DROPS_E, &drops_remainder);
                if (drops_remainder > 10) {
                        pr_info("High (>10) drops ecn remainder: %u\n", drops_remainder);
                }
                testbed->drops_ecn = (__force __u16) drops_remainder;
        } else {
                drops = int2fl(testbed->drops_nonecn, DROPS_M, DROPS_E, &drops_remainder);
                if (drops_remainder > 10) {
                        pr_info("High (>10) drops nonecn remainder: %u\n", drops_remainder);
                }
                testbed->drops_nonecn = (__force __u16) drops_remainder;
        }
        return drops;
}

/* add metrics used by traffic analyzer to packet before dispatching */
void testbed_add_metrics(struct sk_buff *skb, struct testbed_metrics *testbed)
{
        struct iphdr *iph;
        struct ethhdr *ethh;
        u32 check;
        u16 drops;
        u16 id;
        u32 qdelay;
        u32 qdelay_remainder;

        ethh = eth_hdr(skb);
        if (ntohs(ethh->h_proto) == ETH_P_IP) {
                iph = ip_hdr(skb);
                id = ntohs(iph->id);
                check = ntohs((__force __be16)iph->check);
                check += id;
                if ((check+1) >> 16) check = (check+1) & 0xffff;

```

```
                        /* queue delay is converted from ns to units of 32 us and encoded as float */
                        qdelay = ((__force __u64)(ktime_get_real_ns() - ktime_to_ns(skb_get_ktime(skb)))) >> 15;
                        qdelay = int2fl(qdelay, QDELAY_M, QDELAY_E, &qdelay_remainder);
                        if (qdelay_remainder > 20) {
                                pr_info("High (>20) queue delay remainder: %u\n", qdelay_remainder);
                        }

                        id = (__force __u16) qdelay;
                        drops = (__force __u16) testbed_get_drops(iph, testbed);
                        id = id | (drops << 11); /* use upper 5 bits in id field to store number of drops before the current packet */

                        check -= id;
                        check += check >> 16; /* adjust carry */
                        iph->id = htons(id);
                        iph->check = (__force __sum16)htons(check);
                }
}
```

Listing 2: C header file used as an API in the schedulers used in the testbed.

5.1.2 Improving the precision of reporting

The initial code I was given for collecting metrics added the queueing delay in number of milliseconds. The default queueing threshold for DualPI2 is 1 ms, meaning all packets with a queue delay above 1 ms should be marked. The queueing delay uses 11 bits of the *identification* field, giving 2048 different combinations.

To be able to get statistics below 1 ms I implemented a floating point encoding for the numbers being reported. The code implemented for this is given in listing 3. For low queueing delays it reports with a precision of 32 us. As can be seen in figure 5.1, without this encoding the queueing delay could either report 0 ms or 1 ms, and as the decimals are cut off, a lot of numbers were reported as 0 ms. Figure 5.2 show the improved reporting where detailed numbers is given.

As a side effect of this also higher queueing delays can be reported, however the precision will be lower. Figure 5.3 shows how the queueing delay was capped at 2047 ms before, but after adding the encoding figure 5.4 shows queueing delay above this. The example test is limited at 1 000 packets due to the TCP buffer being set equal to 1 000 packets.

```
/* we store drops in 5 bits */
#define DROPS_M 2
#define DROPS_E 3

```

```
/* we store queue length in 11 bits */
#define QDELAY_M 7
#define QDELAY_E 4

/* Decode float value
 *
 * fl: Float value
 * m_b: Number of mantissa bits
 * e_b: Number of exponent bits
 */
u32 fl2int(u32 fl, u32 m_b, u32 e_b)
{
    const u32 m_max = 1 << m_b;

    fl &= ((m_max << e_b) - 1);

    if (fl < (m_max << 1)) {
        return fl;
    } else {
        return (((fl & (m_max - 1)) + m_max) << ((fl >> m_b) - 1));
    }
}

/* Encode integer value as float value
 * The value will be rounded down if needed
 *
 * val: Value to convert into a float
 * m_b: Number of mantissa bits
 * e_b: Number of exponent bits
 * r: Variable where the remainder will be stored
 */
u32 int2fl(u32 val, u32 m_b, u32 e_b, u32 *r)
{
    u32 len, exponent, mantissa;
    const u32 max_e = (1 << e_b) - 1;
    const u32 max_m = (1 << m_b) - 1;
    const u32 max_fl = ((max_m << 1) + 1) << (max_e - 1);
    *r = 0;

    if (val < (1 << (m_b + 1))) {
        /* possibly only first exponent included, no encoding needed */
        return val;
    }

    if (val >= max_fl) {
        /* avoid overflow */
        *r = val - max_fl;
        return (1 << (m_b + e_b)) - 1;
    }

    /* number of bits without leading 1 */
    len = (sizeof(u32) * 8) - __builtin_clz(val) - 1;

    exponent = len - m_b;
    mantissa = (val >> exponent) & ((1 << m_b) - 1);
    *r = val & ((1 << exponent) - 1);

```

```
    return ((exponent + 1) << m_b) | mantissa;
}
```

Listing 3: C header file for encoding/decoding queueing delay and drop numbers.

5.1.3 Drop statistics

When a packet is dropped in the scheduler, two counters are kept representing the number of drops. One for non-ECN packets dropped, and one for ECN capable packets dropped, i.e. a packet with ECT(0), ECT(1) or CE.

On dequeue the number of drops not yet reported will be added as a metric in the packet. The counter this packet belongs to will be used. The counter is then decreased so it will not report the same drop multiple times.

When analyzing the traffic how many packets before the current packet was dropped can be seen.

5.2 Saving the metrics

When running a test, a program is run in the background capturing the traffic going out to the clients.[1] This program decodes the metrics added by the AQM to the packets, and stores data over each sample period specified when running the test.

The program stores files that is later used to plot and derive more statistics from. E.g. the queueing delay is reported for each sample by the number of packets observed in each of the 2048 different combinations of queueing delay that can be reported. Also statistics for each flow is saved so detailed per-flow statistics can be generated.

[1] Available at https://github.com/henrist/aqmt/blob/aef08aa4a8140d28e2689d2be10989c5e96a737a/aqmt/ta/analyzer.cpp. The program contains derived work from an older testbed.

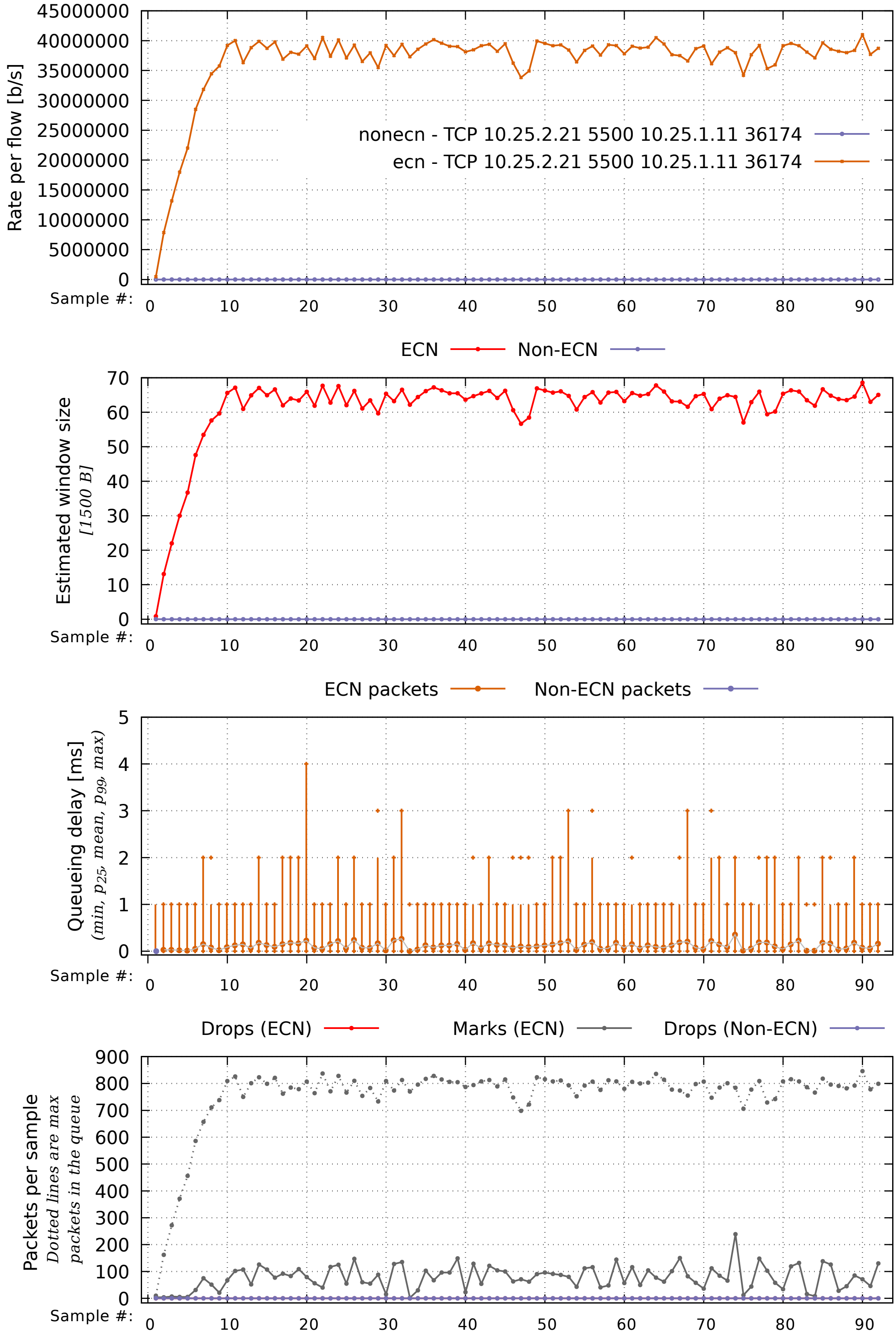


Figure 5.1: Testing lower values of queueing delay - using previous integer version.

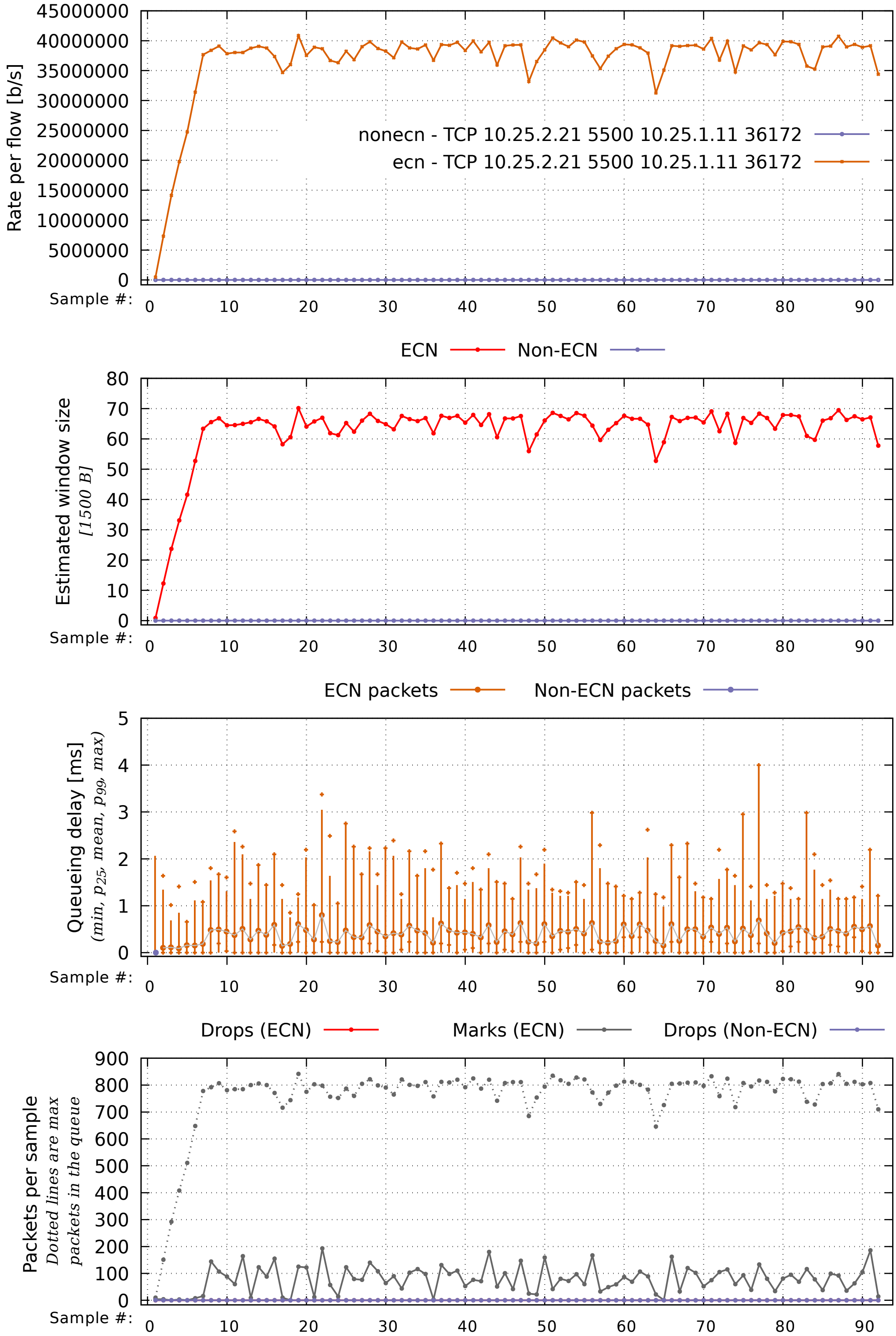


Figure 5.2: Testing lower values of queueing delay - using improved floating point version.

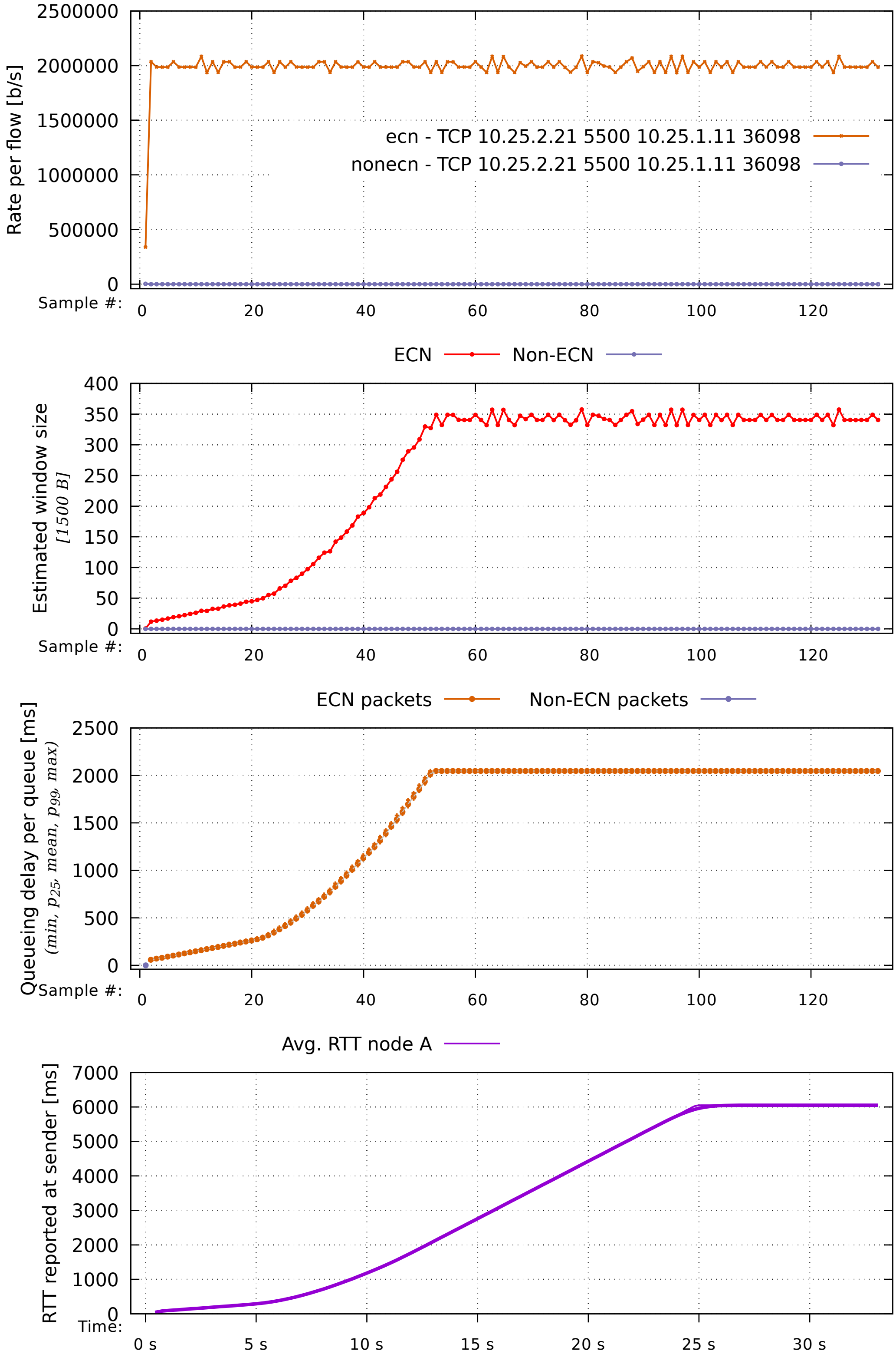


Figure 5.3: Testing high values of queueing delay - using previous integer version.

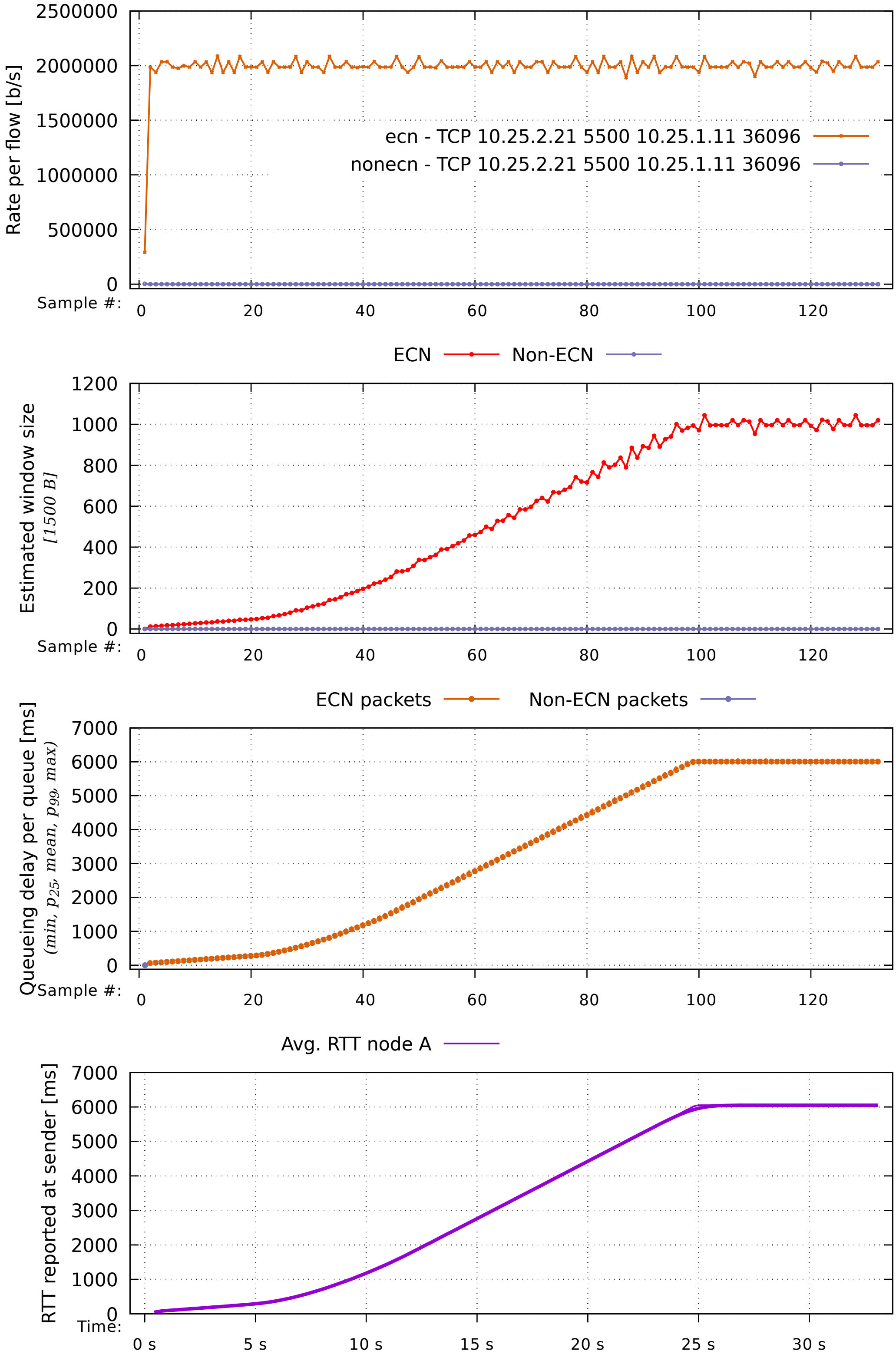


Figure 5.4: Testing high values of queueing delay - using improved floating point version.

Chapter 6

Test framework

A contribution by this thesis is a framework that can be used to test and compare different AQMs. I've called it *Test framework for AQMs* and have released it on GitHub.[1] Part of the source code is given in appendix A.4 for further references. The project consists of approx. 9 000 lines of code. The framework itself is mainly written in Python, but uses several bash scripts and additional compiled programs written in C++. It combines all tools for setting up the testbed, generating traffic, collecting results and plotting results.

The main parts of the framework consists of:

- Tools for constructing all the test parameters and initiating a test.
- Modification of the network configuration.
- Traffic capturing/analysis.
- Analyzing the raw test results.
- Plotting the results.

6.1 Building test definitions

Listing 4 shows a minimal example of how the test framework can be used.

- The example will build a test tree of all the parameters resulting in 18 different tests.
- Each test consists of two flows running greedy (see section 7.1), one using normal CUBIC and one using CUBIC with ECN.
- Each test collects minimum 50 samples and uses 250 ms sample time. The framework actually runs the test a bit longer to let the test stabilize. This can be further customized.

[1] https://github.com/henrist/aqmt

- The test is saved to *results/example* folder and will contain a html file for easy overview of the test.
- A plot comparing the tests will be generated. By default all tests also is plotted individually.

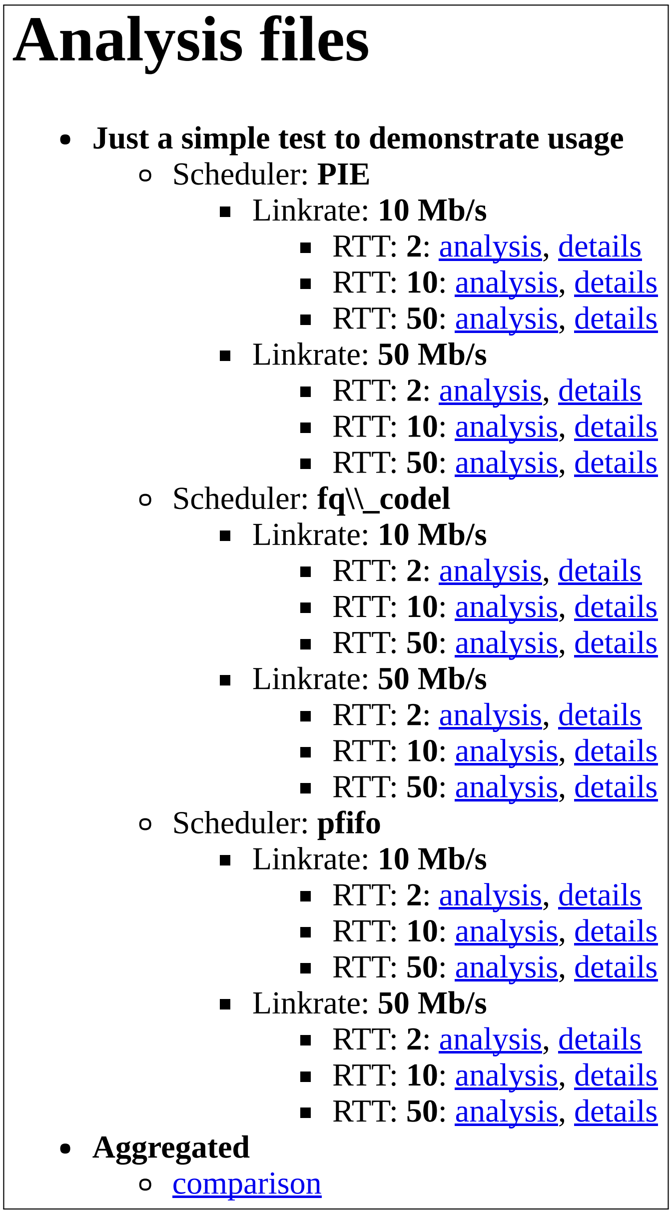
This example uses a high-level abstraction above the framework which wire the different parts of the framework together for easier use. One might also use only part of the framework directly for more control of it.

Listing 13 shows the code that takes such test definitions and transforms it into an actual test using the other components provided by the framework.

```
#!/usr/bin/env python3
#
# This is a very simple example of how to use the
# AQM test framework.
#

import sys

from aqmt import Testbed, TestEnv, run_test, steps
from aqmt.plot import collection_components, flow_components
from aqmt.traffic import greedy


def test(result_folder):

    def my_test(testcase):
        testcase.traffic(greedy, node='a', tag='CUBIC')
        testcase.traffic(greedy, node='b', tag='ECN-CUBIC')

    testbed = Testbed()
    testbed.ta_samples = 50
    testbed.ta_delay = 250

    testbed.cc('a', 'cubic', testbed.ECN_ALLOW)
    testbed.cc('b', 'cubic', testbed.ECN_INITIATE)

    run_test(
        folder=result_folder,
        title='Just a simple test to demonstrate usage',
        testenv=TestEnv(testbed),
        steps=(
            steps.html_index(),
            steps.plot_compare(),
            steps.branch_sched([
                # tag, title, name, params
                ('pie', 'PIE', 'pie', 'ecn'),
                ('fq_codel', 'fq\\\\_codel', 'fq_codel', ''),
                ('pfifo', 'pfifo', 'pfifo', ''),
            ]),
            steps.branch_bitrate([
                10,
                50,
```

```
            ]),
            steps.branch_rtt([
                2,
                10,
                50,
            ], title='%d'),
            my_test,
        )
    )

if __name__ == '__main__':
    test("results/example")
```

Listing 4: Simple example of how the framework is used.



Analysis files

- **Just a simple test to demonstrate usage**
  - Scheduler: **PIE**
    - Linkrate: **10 Mb/s**
      - RTT: **2**: analysis, details
      - RTT: **10**: analysis, details
      - RTT: **50**: analysis, details
    - Linkrate: **50 Mb/s**
      - RTT: **2**: analysis, details
      - RTT: **10**: analysis, details
      - RTT: **50**: analysis, details
  - Scheduler: **fq_codel**
    - Linkrate: **10 Mb/s**
      - RTT: **2**: analysis, details
      - RTT: **10**: analysis, details
      - RTT: **50**: analysis, details
    - Linkrate: **50 Mb/s**
      - RTT: **2**: analysis, details
      - RTT: **10**: analysis, details
      - RTT: **50**: analysis, details
  - Scheduler: **pfifo**
    - Linkrate: **10 Mb/s**
      - RTT: **2**: analysis, details
      - RTT: **10**: analysis, details
      - RTT: **50**: analysis, details
    - Linkrate: **50 Mb/s**
      - RTT: **2**: analysis, details
      - RTT: **10**: analysis, details
      - RTT: **50**: analysis, details
- **Aggregated**
  - comparison

Figure 6.1: HTML file generated for the test.

6.1.1 Building test tree

The core of test definitions is the *steps* provided to the *run_test* function. These acts as middlewares that can create branches in the tree. Python generators[2] is used to achieve this. Each step either yield a empty value simply passing control to next middleware, or it can yield one or multiple objects defining a node in the tree. For each yield control is passed to next middleware. The middlewares mutate the test definition, so that when the actual test function is reached as the last step, it will

[2] https://wiki.python.org/moin/Generators

use the previous defined parameters.

Listing 5 shows how this can be implemented. The framework includes a few usefull middleware creators that accept the parameters in a functional style. See listing 14 for the included middlewares that can be used out of the box.

```
def branch_rtt(testdef):
    for rtt in [10, 50]:
        testdef.testenv.testbed.rtt_servera = rtt
        testdef.testenv.testbed.rtt_serverb = rtt
        yield {
            'tag': 'rtt-%d' % rtt,
            'title': rtt,
            'titlelabel': 'RTT',
        }
```

Listing 5: A over-simlified middleware that causes a branch in the tree for testing RTT 10 and RTT 50.

6.2 Built in metrics

The framework can be extended to provide further metrics. Most likely the interesting metrics is already provided as ready to be graphed:

- Utilization for non-ECN vs ECN traffic.
- Utilization per flow, optionally grouped by a specified identifier.
- Queueing delay.
- Estimated window sizes.
- Drop and mark numbers.
- Window ratio between non-ECN and ECN flows.
- CPU usage statistics.
- Actual RTT observed at the sender.

6.3 Other features

- The test structure saved to disk can be reconstructed through the meta files stored with the tests. This is actually done when using the default plotting.
- The idle time in the tests can be moved in time after the test is run. This is usefull when running a huge test, and later discovering the idle time was not enough to let the test stabilize, causing the comparison plots to be unstable. Simply rerunning the test definition on top of the existing test data with a reanalyze flag after increasing the idle time will generate new derived data.

- The included plot functions is optional to use. Custom plottings can easily be added through the use of custom middlewares, or adding custom plot functions to the existing plot framework. E.g. two plugins for collecting *dstat*[3] statistics and plotting this, and collecting RTT reported from the sender side is included, implementing such features.

- The test structure can be restructured after the test is done, e.g. to group parameteres differently. See listing 17 for more details about this. The function in listing 18 shows how the initial tree is built by using the test results stored to disk.

- The traffic generators are simply functions you provide to the framework, and custom ones can be created.

- By having open sourced the framework the idea is that it will be further improved and be used for later work.

- A lot of utility scripts are provided to monitor the state of the testbed. E.g. watching statistics from the qdiscs, graphing the current traffic rates, inspecting statistics from *ss*, monitoring the interfaces for drop status, and more.

See the GitHub page for a more complete walkthrough for how the framework works and how to get started using it.

[3] http://dag.wiee.rs/home-made/dstat/

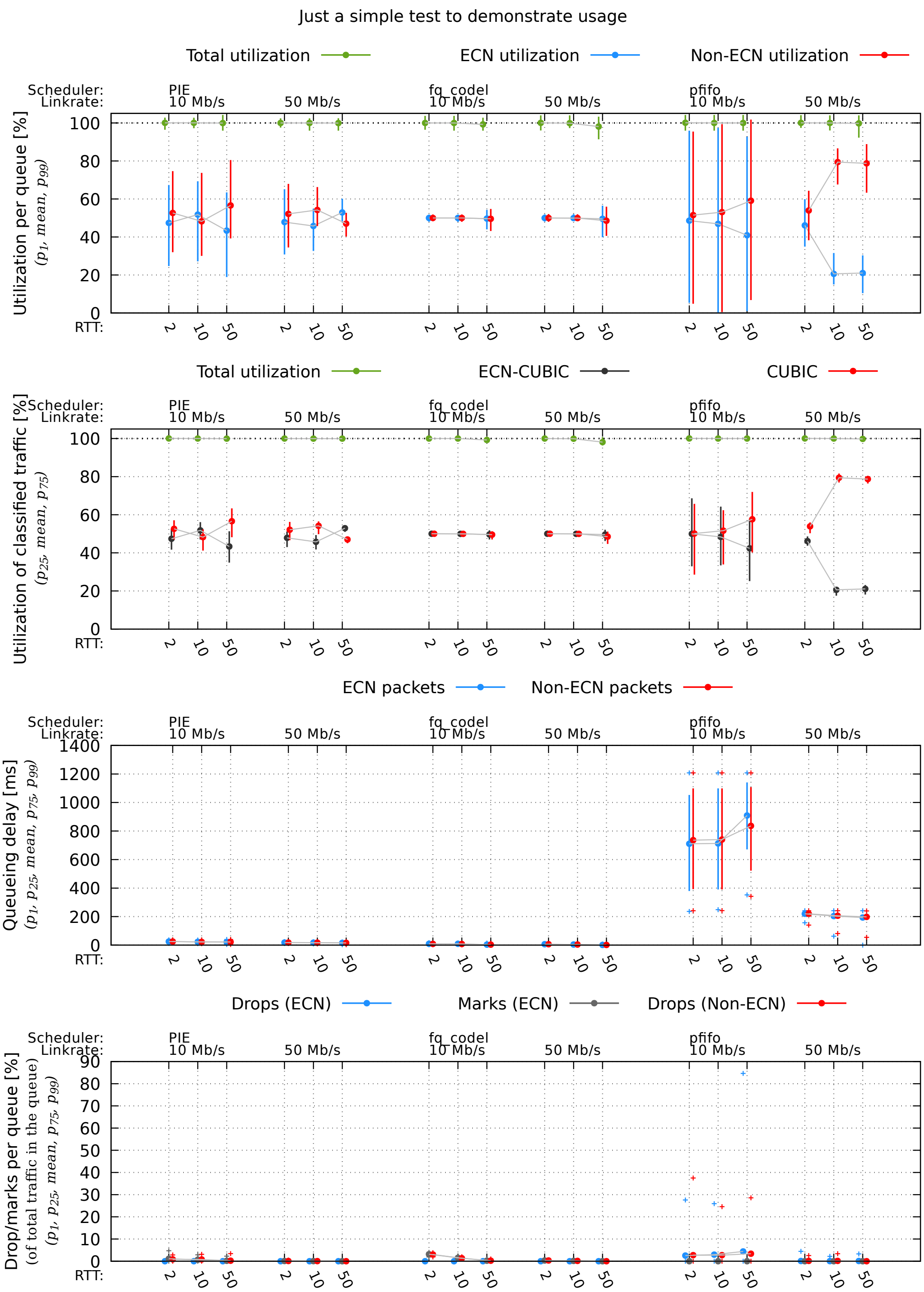


Figure 6.2: Default comparison plot

Chapter 7

Traffic generation tools

7.1 Greedy

Greedy is a client/server test application written by me for this thesis in order to generate TCP traffic, and has also been open sourced.[1] The goal of greedy is to fill available buffer space in Linux so that data can always be transmitted as long as the kernel is ready. This way application behaviour should likely not influence the test results.

One of the ways greedy manages to fill the buffers is not to use blocking writes. The first version used blocking writes which caused bursty behaviour when there were no room in the buffer and all packets where in flight. The kernel would block while freeing up buffer space (receiving ACKs), and then on next unblock a lot of packets would get queued and instantly sent because there were room in the congestion window. The non-blocking version manages to buffer small amount of data as long as there is free space in the buffer.

The application also emits information from *tcp_info* structure available through the kernel API yielding numbers such as buffers, ECN flags, window size, lost packets, packets in flight and more, similar to what *ss* command gives. However, this is currently only used for visual monitoring, not for analysis.

Most of the tests in this thesis using TCP is done with greedy. The source code for greedy is given in listing 8.

7.2 SSH and SCP

Initially SCP over SSH was used to generate data. However while testing various parameter it showed that this was not reliable at all. SSH is a multiplexing application having its own flow control and window implementation limiting performance.[24] Considering it also adds additional overhead caused by encryption, it should be no surprise it should avoided in order to get reliable results.

[1] https://github.com/henrist/greedy

SCP should in all cases be avoided as it need to read/write from/to disk and it might cause iowait and blocking.

7.3 iperf2 for TCP

Support for using iperf2 in the test is added. But in the final tests I have only used *greedy*. Testing has shown that iperf2 is not reliable and stable with a high BDP. iperf2 fails to keep the maximum congestion window allowed by the TCP kernel memory settings.

7.4 iperf2 for UDP

For generating UDP traffic I have used iperf2. iperf2 also takes an argument for the *TOS*-field, which can be used to set the ECN flags and control which queue it goes into.

7.4.1 A note about iperf3 for UDP traffic

Initially iperf3 was used, but as of currently the timer implementation in iperf3 only sends UDP data every 100 ms [2], causing extremely bursty behaviour. The bursts caused a on/off pattern in the AQM. iperf2 does not have this problem, so I have sticked to it.

7.5 Comparing traffic generators

To evaluate the different traffic generators I have run a few tests to see how they perform when reaching the limit of their congestion window. The parameters set for this test:

- The TCP kernel buffer size is set to the default value giving a maximum window size of 965 full packets, as discussed later in section 9.3.2.
- Bitrate is set to 200 Mbit/s.
- Base RTT is set to 50 ms.
- CUBIC is used as congestion control.
- pfifo is used as scheduler. In this test example no drops or marks is occuring.

The results from this test is shown in figure 7.1 and can be summarized as following:

greedy Due to constantly trying to fill the kernel TCP buffer, greedy maintains a high congestion window and gets a high utilization.

[2] https://github.com/esnet/iperf/pull/460

SCP and SSH SCP and SSH gets the lowest utilization, as well as lowest queueing delay. In this result SCP and SSH-only gives similar results, but using higher bandwidth will cause greater iowait using SCP causing unstability.

iperf2 and netcat The two give similar results. The window seems to not get above 820 packets in average. The queueing delay is also a lot higher than the others, and quite unstable, even though the utilization is lower. This is most likely because the kernel runs out of buffer space while all packets are in flight, causing the TCP application to sleep, and suddenly being able to buffer lots of packets causing a burst. This was discussed when in the presentation of greedy in section 7.1.

Greedy clearly outperforms the others. One might argue this isn't a realistic way of filling the TCP buffers, however this avoids having another factor that might cause errors in the testing. Instability of TCP senders is not of interest for the questions I am exploring.

From testing higher window sizes by increasing the TCP memory, testing shows that SSH and SCP is limited to a window size of approx. 1 500 packets and refuses to buffer more data. This is probably due to the multiplexing in SSH and the internal window it maintains.

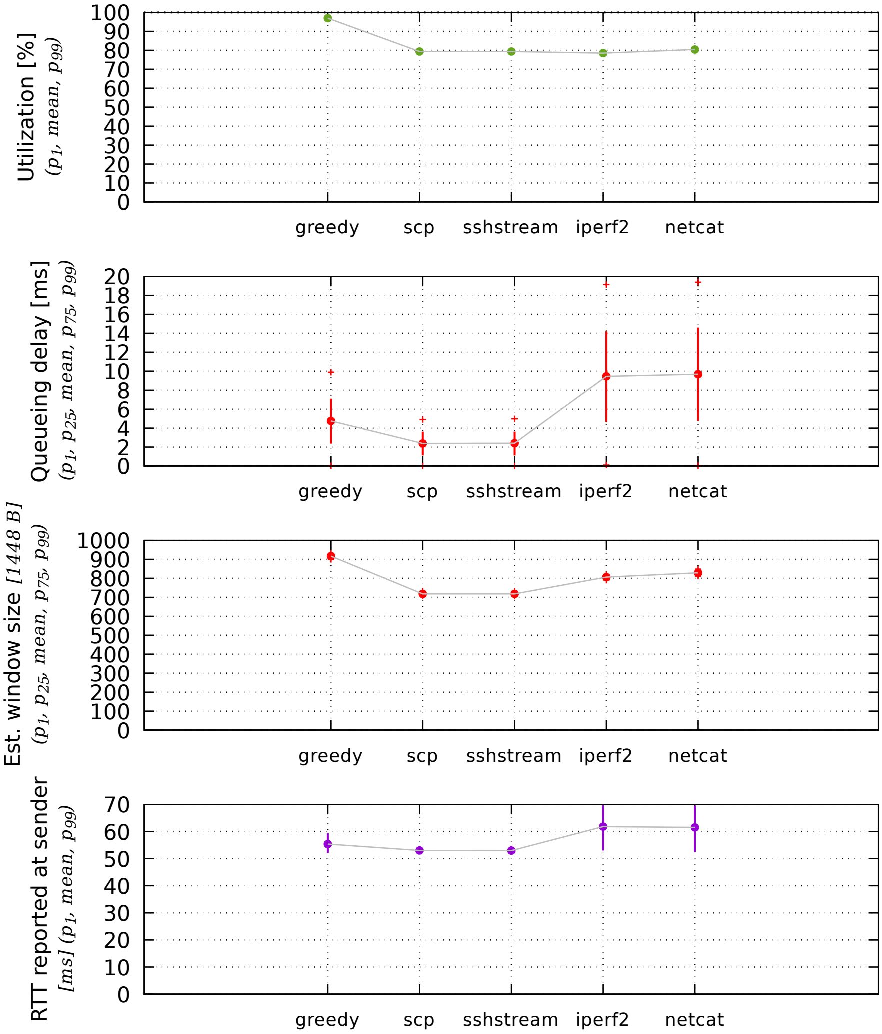


Figure 7.1: Comparing traffic generators.

Chapter 8

Virtualized testing with Docker

8.1 Introduction to Docker and containers

As part of this thesis I wanted to investigate whether I could run tests without needing a full physical testbed. As I had experience with Docker[27], it came up as a possibility to investigate. Docker is a platform used to manage containers running on a host system. It acts as an abstraction layer on top of the operating system to manage containers. Containers are not fully virtual machines, such as *QEMU*[1] or *KVM*[2], but uses namespacing capabilities of the host system to form isolation between containers. Even though Docker can be used on other systems such as Windows, only Linux is in scope for this thesis.

A side effect of using Docker and isolating all the test infrastructure the experiments are more easily reproducable. Instead of manually configuring a physical testbed, we can use one definition everywhere.

A concern using Docker is that all containers and processes running inside it shares the host kernel, as well as running on the same CPU. In Linux only some configuration options are namespaced and hence can be specified differently for the containers. E.g. the buffer limits for TCP connections are not exported to the container. The containers can neither manage kernel modules, and is using what the host system has loaded.

The goal of using Docker is to show that it can be used as an alternative to a physical testbed.

[1] http://www.qemu.org/
[2] https://www.linux-kvm.org/

8.2 Defining the containers and network topology for Docker

To define the testbed containers with Docker I use a tool called Docker Compose[3]. The tool allows us to define containers and network in a configuration file which is used to provision the containers, networking and other needed features.

My definition used for defining a similar topology as with the physical testbed is given in listing 11. As containers run in a seperate file system, we also mount a few directories to be available inside the containers.

8.3 Networking in containers

Each container is put in a namespaced network stack forming isolation. Docker manages all this for us, but similar could have been done manully using the *ip-netns - process network namespace management*[4] commands.

As the network is isolated, it cannot communicate outside its isolation. To resolve this Docker manages a *virtual Ethernet interface* which acts as a connection between two namespaced network. This can be seen similar to a physical cable connecting two machines. For each network in a container there is:

- A virtual Ethernet interface with one end acting as an interface inside the container, and the other end connected to a bridge on the host system.

Containers can communicate with each other by being connected to the same bridge (same network in Docker context). To match the topology there are four networks:

- Management network. All containers are connected to it. The management network allows us to control the machines without using the interfaces under testing.
- Clients network. All clients and the AQM machine is connected. This is the network that acts as a bottleneck for data from the AQM machine.
- Server A network. Only the AQM and server A is connected.
- Server B network. Only the AQM and server B is connected.

[3] https://docs.docker.com/compose/overview/
[4] http://man7.org/linux/man-pages/man8/ip-netns.8.html

8.3.1 Initializing networks in the container

When the containers are started they will perform the neccessary initialization to set up the network. The script performing this is given in listing 10.

- Disable segmentation offloading. See Section 9.1.
- Add static routes to the other machines through the AQM to allow them machines to talk to each other using the AQM as a router.
- Set *txqueuelen* interface option to the normal 1 000. As the interfaces are virtual, the default is to not allow any queueing in the interface.
- Add static ARP entries to the gateway to avoid APR lookups as seen in section 9.2.
- Reset queuing disciplines
- Collect information about the network interfaces name etc. for later use.

8.3.2 Setting congestion control and ECN feature

The *net.ipv4.tcp_congestion_control* option cannot be changed from inside a container, and is global for all traffic on the host. However, as of iproute2 v4.6 the *congctl* option was added as per route option. In addition, *ecn* can be enabled per route the same way. The *configure_host_cc* function in listing 15 shows how this is done.

Chapter 9

Pitfalls

9.1 Segmentation offloading

Segmentation offloading lets the kernel move some TCP/IP processing to the network card, giving a performance improvement. A side effect of offloading the packets is they are also combined into larger segments. A 1500 byte segment might be combined with other segments causing larger packets.

Offloading makes it confusing to inspect packets, and packets handled by the AQM will actually be grouped into one packet, causing wrong behaviour. Offloading also cause different testing results depending on the underlying hardware.

The following offloading features have been disabled in all tests:

- *gso* - generic segmentation offload
- *gro* - generic receive offload
- *tso* - tcp segmentation offload

The segmentation offloading is changed by using the *ethtool* utility, as shown in listing 6.

```
#!/bin/bash
iface=eth0
ethtool -K $iface gro off
ethtool -K $iface gso off
ethtool -K $iface tso off
```

Listing 6: Shell script to disable offloading.

9.2 ARP requests causing silent periods

During testing I encountered silent periods in the tests, basicly time where there were no traffic. Using *wireshark* I identified that there were ARP[9] requests going on at the same time.

ARP requests looks up which ethernet address to send traffic for a given IP address. Requests are sent out on the network, and a neighbour that wants to receive this traffic announces itself. Traffic for a specific IP is then sent to that ethernet address.

For some reason not investigated, this happened quite often, both on the physical testbed as well as in the virtualized.

This was resolved by explicitly adding ARP tables for the different machines, as shown in listing 7.

```
#!/bin/bash
source aqmt-vars.sh

mac_clienta=$(ssh $IP_CLIENTA_MGMT "ip l show $IFACE_ON_CLIENTA | grep ether | awk '{ print \$2 }'")
mac_clientb=$(ssh $IP_CLIENTB_MGMT "ip l show $IFACE_ON_CLIENTB | grep ether | awk '{ print \$2 }'")
mac_servera=$(ssh $IP_SERVERA_MGMT "ip l show $IFACE_ON_SERVERA | grep ether | awk '{ print \$2 }'")
mac_serverb=$(ssh $IP_SERVERB_MGMT "ip l show $IFACE_ON_SERVERB | grep ether | awk '{ print \$2 }'")

mac_aqm_clients=$(ip l show $IFACE_CLIENTS | grep ether | awk '{ print $2 }')
mac_aqm_servera=$(ip l show $IFACE_SERVERA | grep ether | awk '{ print $2 }')
mac_aqm_serverb=$(ip l show $IFACE_SERVERB | grep ether | awk '{ print $2 }')

# clients -> aqm
ssh root@$IP_CLIENTA_MGMT "arp -i $IFACE_ON_CLIENTA -s $IP_AQM_C $mac_aqm_clients"
ssh root@$IP_CLIENTB_MGMT "arp -i $IFACE_ON_CLIENTB -s $IP_AQM_C $mac_aqm_clients"

# aqm -> clients
sudo arp -i $IFACE_CLIENTS -s $IP_CLIENTA $mac_clienta
sudo arp -i $IFACE_CLIENTS -s $IP_CLIENTB $mac_clientb

# servers -> aqm
ssh root@$IP_SERVERA_MGMT "arp -i $IFACE_ON_SERVERA -s $IP_AQM_SA $mac_aqm_servera"
ssh root@$IP_SERVERB_MGMT "arp -i $IFACE_ON_SERVERB -s $IP_AQM_SB $mac_aqm_serverb"

# aqm -> servers
sudo arp -i $IFACE_SERVERA -s $IP_SERVERA $mac_servera
sudo arp -i $IFACE_SERVERB -s $IP_SERVERB $mac_serverb
```

Listing 7: Shell script to add static entries to the ARP tables.

9.3 Buffer limits testing high BDP

9.3.1 Buffer size for base RTT

As can be seen from table 3.1, having a high RTT requires a larger buffer. *netem* is used to simulate delay on path. The base RTT is split in two, half in each direction. *Netem* needs to buffer all the data that sits through this intended delay.

The default limits on Linux is 1000 packets of queueing. As offloading is disabled, each packet (or more correctly, each segment) contains 1448 bytes of data, equaling 1,38 MiB for 1000 packets. Given the RTT and limit, when this limit will be exceeded can be calculated:

$$\frac{packets \times (8 \times 1\ 448)\ \text{b}}{rtt\ \text{s}} = \frac{1\ 000 \times (8 \times 1\ 448)\ \text{b}}{0.05\ \text{s}} = 220\ \text{Mbit/s}$$

Notice this is the rate of the application data, not the link rate.

Exceeding rate will cause drops that is not seen by the AQM. When adding the *netem* qdisc this has to be taken into consideration and highter limits applied if needed. This is done by adding a *limit* option specifying the limit in packets.

9.3.2 Kernel TCP memory limits

Linux has a limit to how much buffer space it has allocated to a TCP connection. The buffer has to hold on to all packets since the last consecutive received ACK in case it has to retransmit data. The TCP window size is hence limited to the buffer allocated in the kernel.

The buffer sizes can be controlled through sysctl changing the *net.ipv4.tcp_rmem* (for receive buffer) and *net.ipv4.tcp_wmem* (for send buffer) settings. In addition to holding on to packets not yet ACKed, the kernel will buffer data from the application that will be ready for transmission when the TCP window allows it.

By default on Linux, the receiving buffer is max 6 MiB and the send buffer is max 4 MiB. [1] Through testing I have noticed that this limits the window in number of packets as such:

- *tcp_rmem* has to be double the maximum window times MSS.
- *tcp_wmem* has to be tripple the maximum window times MSS.

Given a MSS of 1 448 bytes, the default maximum values yield:

- *tcp_rmem* gives a maximum window size of $\frac{tcp\_rmem}{1\ 448\ \text{bytes} \times 2} = \frac{6\ \text{MiB}}{1\ 448\ \text{bytes} \times 2} = 2\ 172$ packets
- *tcp_wmem* gives a maximum window size of $\frac{tcp\_rmem}{1\ 448\ \text{bytes} \times 3} = \frac{6\ \text{MiB}}{1\ 448\ \text{bytes} \times 3} = 965$ packets

[1] https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/Documentation/networking/ip-sysctl.txt

If this buffer is filled up, and all packets are in flight, no data will be ready in the kernel for transmission when ACKs are received.

Note that when using the *ss* command the congestion window will actually be higher, but it will not be allowed to send more packets even though the window is higher. By looking at the *unacked* number you will see number of packets in flight.

9.4 Implicit delay at low bandwidth

When simulating a slow connection, e.g. by using *netem*, it will give a higher noticeable base RTT due to the fact that the rate limiting has to block packets to keeping down the rate.

Using a bitrate of 2 Mbit/s for 1500 bytes gives the following propagation delay:

$$\frac{\textit{packet size}\ \text{b}}{\textit{bitrate}\ \text{b/s}} = \frac{12\ 000\ \text{b}}{2\ 000\ 000\ \text{b/s}} = 0.006\ \text{s} = 6\ \text{ms}$$

Figure 9.1 shows an example of this. As can be seen the reported RTT by the server is 20 ms, while the average queueing delay in DualPI2 is between 3 ms or 4 ms. The difference not seen by DualPI2 is 6 ms.

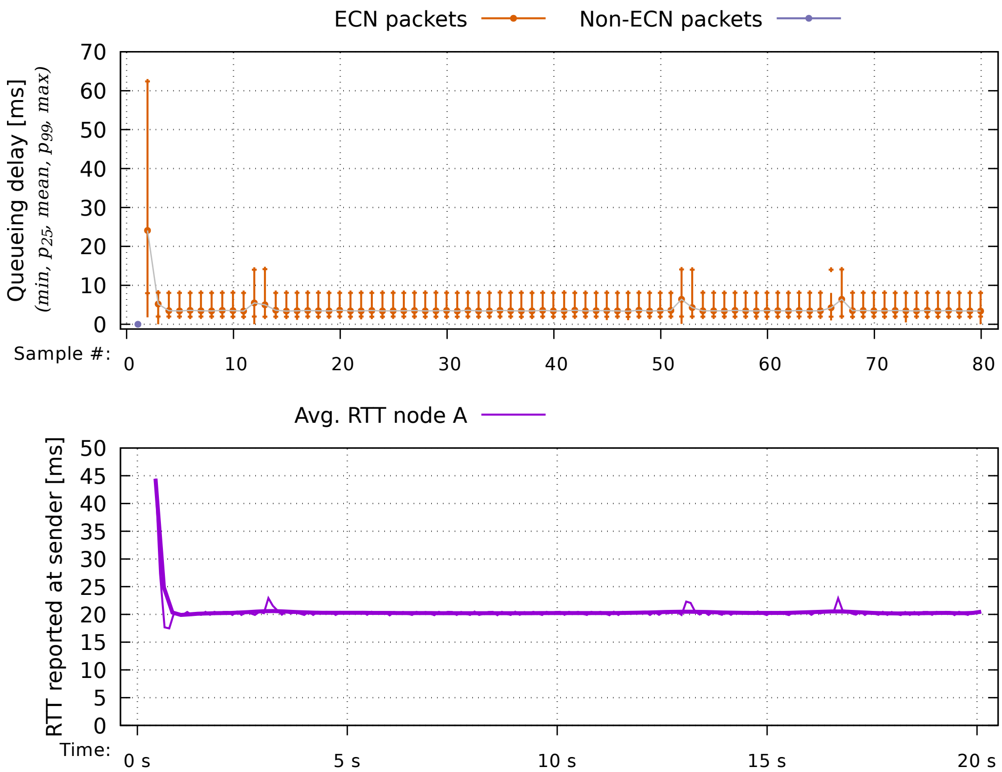


Figure 9.1: DualPI2 as AQM. 2 Mbit/s link rate. 10 ms base RTT. One DCTCP flow.

Chapter 10

Improving the DualPI2 implementation

During working with my thesis I have also contibuted to improving the DualPI2 reference implementation.

10.1 Moving the drop from enqueue to dequeue

The paper describing DualPI2 specifies the dropping to be done at dequeue. However, the reference implementation used in the paper for evaluation, and which I have used, have the drop applied when packets are enqueued.

I have moved the drop to be applied on dequeue as specified, and the tests in this paper reflects this change.

Part III

Evaluation

Chapter 11

Default parameters

11.1 Default parameters unless otherwise noted

DualPI2 limit 1000p target 15.0ms tupdate 16.0ms alpha 5 beta 50 dc_dualq dc_ecn k 2 l_drop 100 l_thresh 1.0ms t_shift 30.0ms

PIE limit 1000p target 15.0ms tupdate 16.0ms alpha 1 beta 10 ecn

Chapter 12

Overload in DualPI2

12.1 Response to simple overloading

The very simplest overload is running overload alone, without any other traffic. I have compared PIE vs DualPI2 for these tests. The code for running the tests are given in listing 19.

Testing high link rates, figure 12.1 shows PIE generates a on/off pattern trying to handle the overload when the buffer size is high. Using a low buffer size, shown in figure 12.2, the problem is not so visible. However, with a smaller buffer, the queue delay is limited because excessive traffic is tail dropped, causing a slow response by the PI controller. This is also visible using DualPI2, as seen in figure 12.3. DualPI2 do not have the issue with on/off pattern, see figure 12.4.

Putting overload in the L4S of DualPI2 with similar parameters, see figure 12.5, shows the point where overload handling is being effective when the probability reaches 100 % marking for the L4S, causing drops instead, which causes the delay to decrease.

From these results, we can also see that DualPI2 is quicker than PIE to respond to high queueing, while it uses approx. 1 second to linearly reduce 10 % of drop probability when the queue is empty.

The results show that having a small buffer when being overloaded leads to a slow handling of the feedback because the PI controller is responding slowly and no packets are being dropped.

12.2 Impact of overload in existing traffic

Overload alone is interesting in itself, but introducing other traffic at the same time shows a more realistic case where other flows are affected by it. DualPI2 is targeting no poorer performance than PIE, which makes it appropriate to use as a reference.

To evaluate overload with mixed traffic we run different combinations of greedy TCP traffic and introducing overload by running a UDP flow at constant rate. To test how and when the overload handling in DualPI2 takes effect, we run the test over a variety of overload rates. We also run UDP flows below the linkrate without generating overload, but the UDP flow is still un-responsive. The complete test script is given in listing 20.

All these tests are run with 10 ms RTT on top of 100 Mbit/s link rate. We test with UDP traffic in the classic queue and comparing it against running UDP in the L4S using ECT(1). When testing PIE we use CUBIC with ECN enabled instead of DCTCP. The comparison plot contains data after the flows has stabilized.

1. Figure 12.6 and 12.7 show UDP traffic in the classic queue.

2. Figure 12.8 and 12.9 show UDP traffic using ECN, going in the same queue for PIE but in the L4S for DualPI2.

Interpreting the results

The statistics shown in the graph helps us explaining what is going on. Because of the amount of different test cases and individual tests (554 to be exact), we go through each one comparing DualPI2 with PIE. Each item in the list represents the UDP queue, number of non-ECN TCP flows and number of ECN-capable TCP flows, same as the plot is ordered:

Non-ECT, 0 vs 1 Common for all plots, we can see that having ECN-traffic with DualPI2, the marking stats is extremely high. This is natural due to the DCTCP algorithm receiving constant feedback.

At approx. 130 Mbit/s UDP the 50 % probability causes the overload mechanism in DualPI2 to kick in. This switches to square dropping the ECN packets similar to the classic queue. Any packets leaving the queue will still be marked.

However, the graph shows no ECN packets are actually being marked. The most likely explanation of this is because of the combination of a priority scheduler and dropping on dequeue. The few ECN packets not being dropped has been prioritied such that their queueing delay is below the threshold for marking.

Also, recall that the drop probability in DualPI2 is squared, so e.g. the drop probability at 120 Mbit/s UDP in PIE of approx. 20 % matches the square of DualPI2 drop probability of approx. 45 %.

Comparing with PIE it seems DualPI2 gets quite similar results. However, before the overload mechanism kicks in more capacity is left for the DCTCP traffic. Also the DCTCP traffic gets much lower latency while the overload is going on. PIE actually gets a lower average queueing delay, however it fails to maintain its actual target, which DualPI2 achieves.

Common for almost all scenarious is that the capacity left for other flows than the UDP flow is very little.

Non-ECT, 1 vs 0 Having all traffic in the classic queue shows almost identical results between DualPI2 and PIE. Only noticable is that DualPI2 keeps a more stable average queueing delay.

Non-ECT, 5 vs 5 Introducing normal TCP traffic in both queues shows how the priority queue gives the ECN traffic more capacity. The RTT plots confirms that the ECN traffic maintains a very low queue.

Looking at the drop probabilities we can see that PIE gets very unstable. Its p1 value stays consistent, while the average and p99 grows. For DualPI2 the p1 value follows the increase of the average.

As we are having more traffic in the test, the overload switch for DualPI2 happens earlier, where we have more data points, so we can clearly see the marking reduces as in the first test. Otherwise, the results are not that far from the first two tests.

ECT(1), 0 vs 1 When overload traffic is sent to the L4S queue things change quite a bit. For utilization, it seems like DualPI2 compares with PIE. The queueing delay is far more stable for DualPI2. In PIE a lot of non-ECN drops is reported, which is due to CUBIC with ECN retransmitting packets without ECN.

Introducing UDP traffic with ECN means there is no random dropping of the packets until the overload mechanism kicks in. Having overload just above the link rate (or at or slightly under with many other flows) will cause the queue to grow, activating the overload mechanism. As such the overload mechanism in these cases kicks in when we send UDP just about the same as the link rate.

As with the previous test, PIE has a very high variation of drop probabilities. This also happens in the next two tests. The result from this test shows that DualPI2 seems to be more stable than PIE in this test.

ECT(1), 1 vs 0 Introducing classic traffic while overloading the L4S queue shows DualPI2 getting very unstable. By looking at the probability plot we can see that during this unstability the probability seems to oscilliate with a on/off overload behaviour. When overload is enabled, so many packets in the L4S queue is dropped that the classic queue reduces, and when overload is disabled, the queue grows and causing the L4S queue to grow when the time shifted priority scheduler reaches its threshold.

Comparing with PIE, the utilization of the TCP traffic in DualPI2 is considerable worse. A possible solution for this that might be worth investigating is to look into whether the time shifted priority scheduler should work otherwise during overload.

ECT(1), 5 vs 5 The results of having both classic and ECN traffic at the same time seems to be comparable with the two previous tests. From the queueing delay and RTT plots we can see that ECN traffic still maintains low delay. The utilization seems to be comparable with the similar test having UDP in the classic queue.

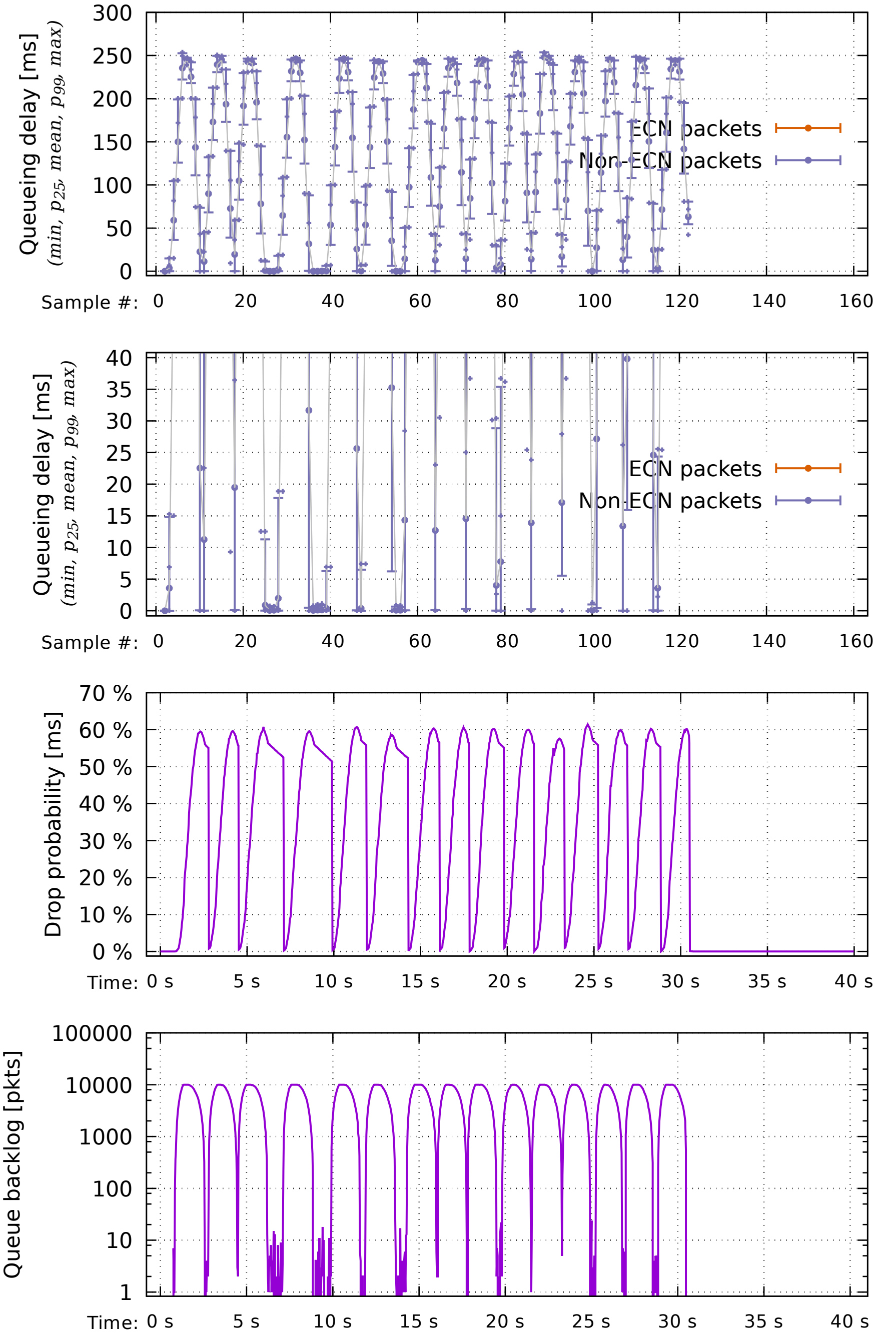


Figure 12.1: PIE as AQM with 10 000 packets limit. 500 Mbit/s link rate. 800 Mbit/s UDP traffic with no ECN. Results with ECN show similar results.

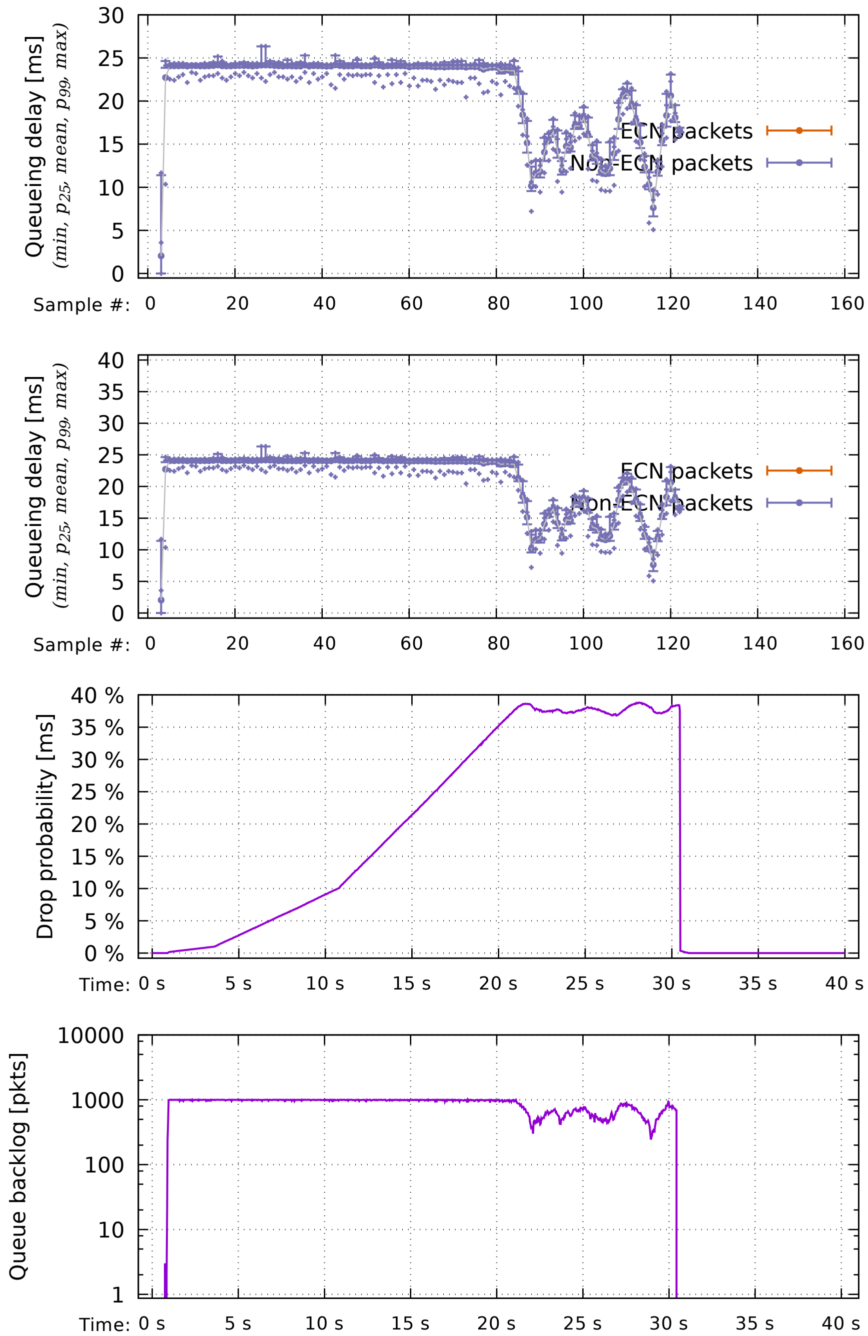


Figure 12.2: PIE as AQM with 1 000 packets limit. 500 Mbit/s link rate. 800 Mbit/s UDP traffic with no ECN. Results with ECN show similar results.

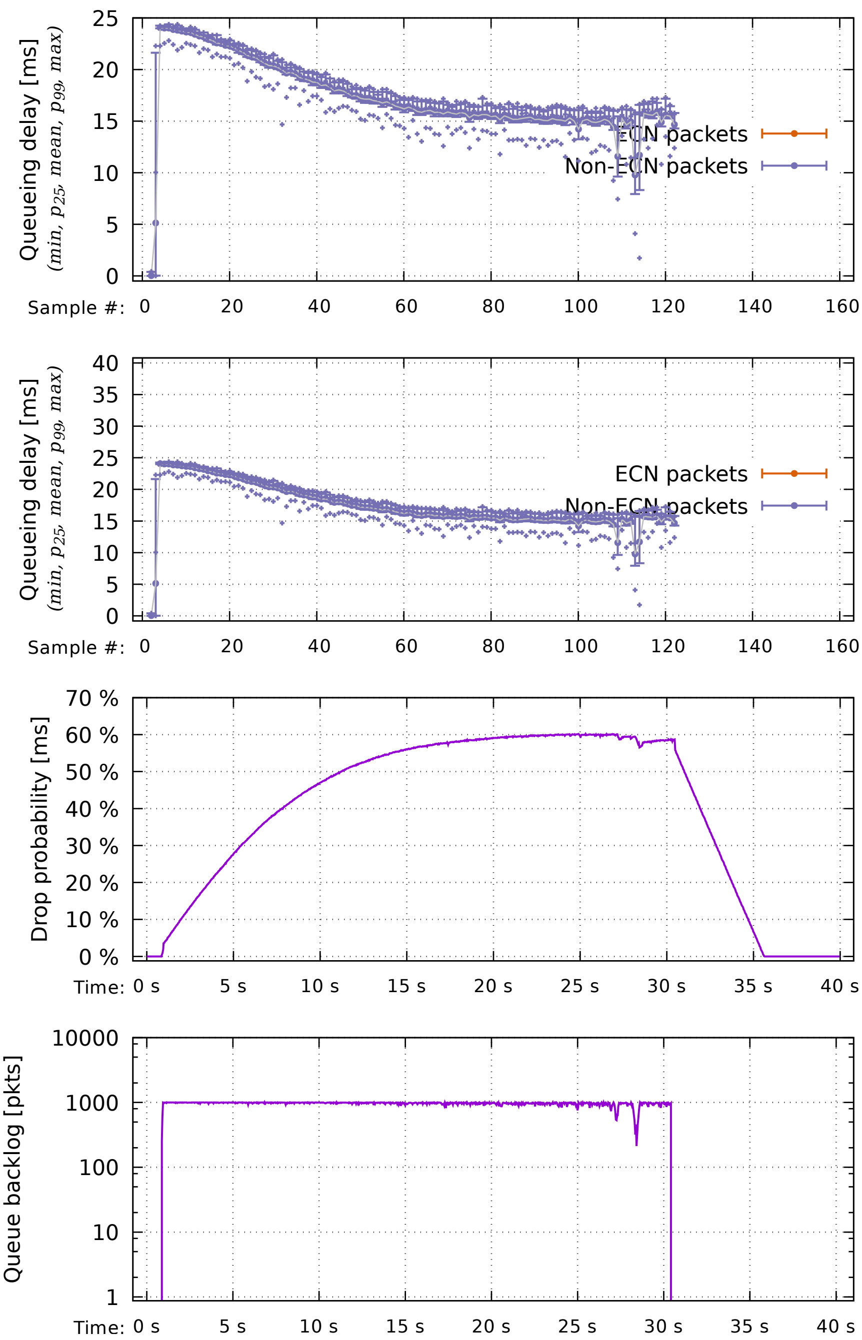


Figure 12.3: DualPI2 as AQM with 1 000 packets limit. 500 Mbit/s link rate. 800 Mbit/s UDP traffic with no ECN.

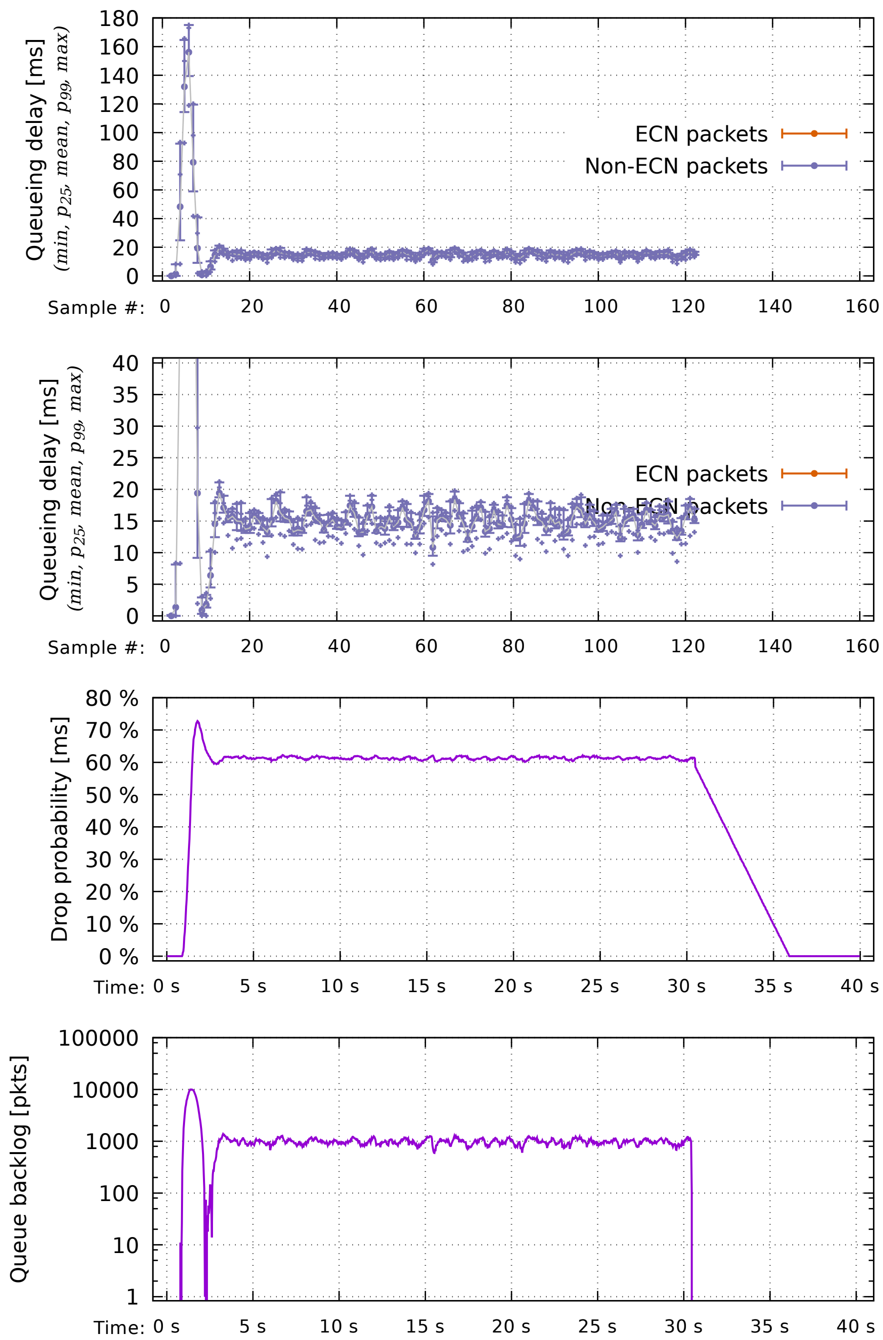


Figure 12.4: DualPI2 as AQM with 10 000 packets limit. 500 Mbit/s link rate. 800 Mbit/s UDP traffic with no ECN.

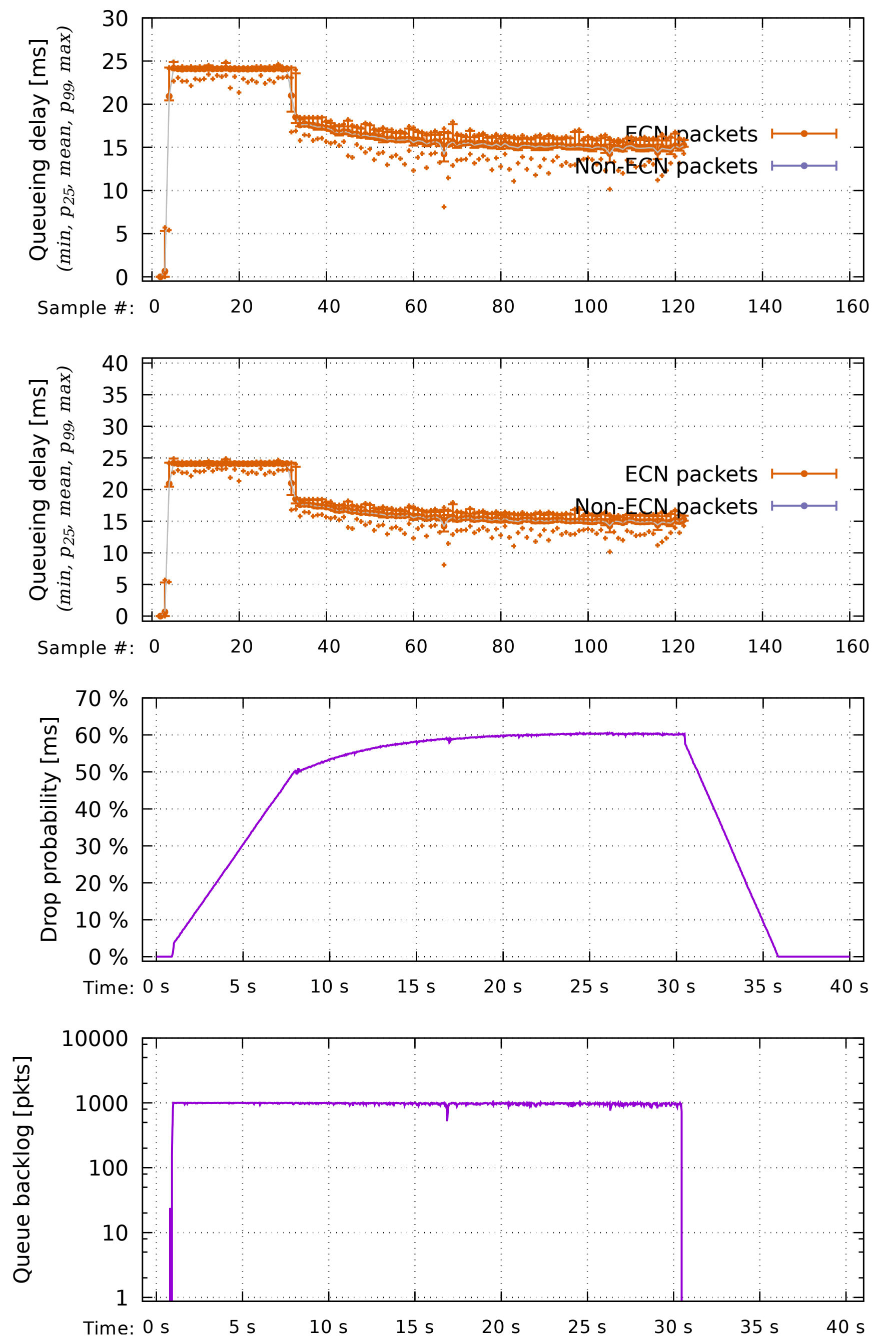


Figure 12.5: DualPI2 as AQM with 1 000 packets limit. 500 Mbit/s link rate. 800 Mbit/s UDP traffic with ECT(1).

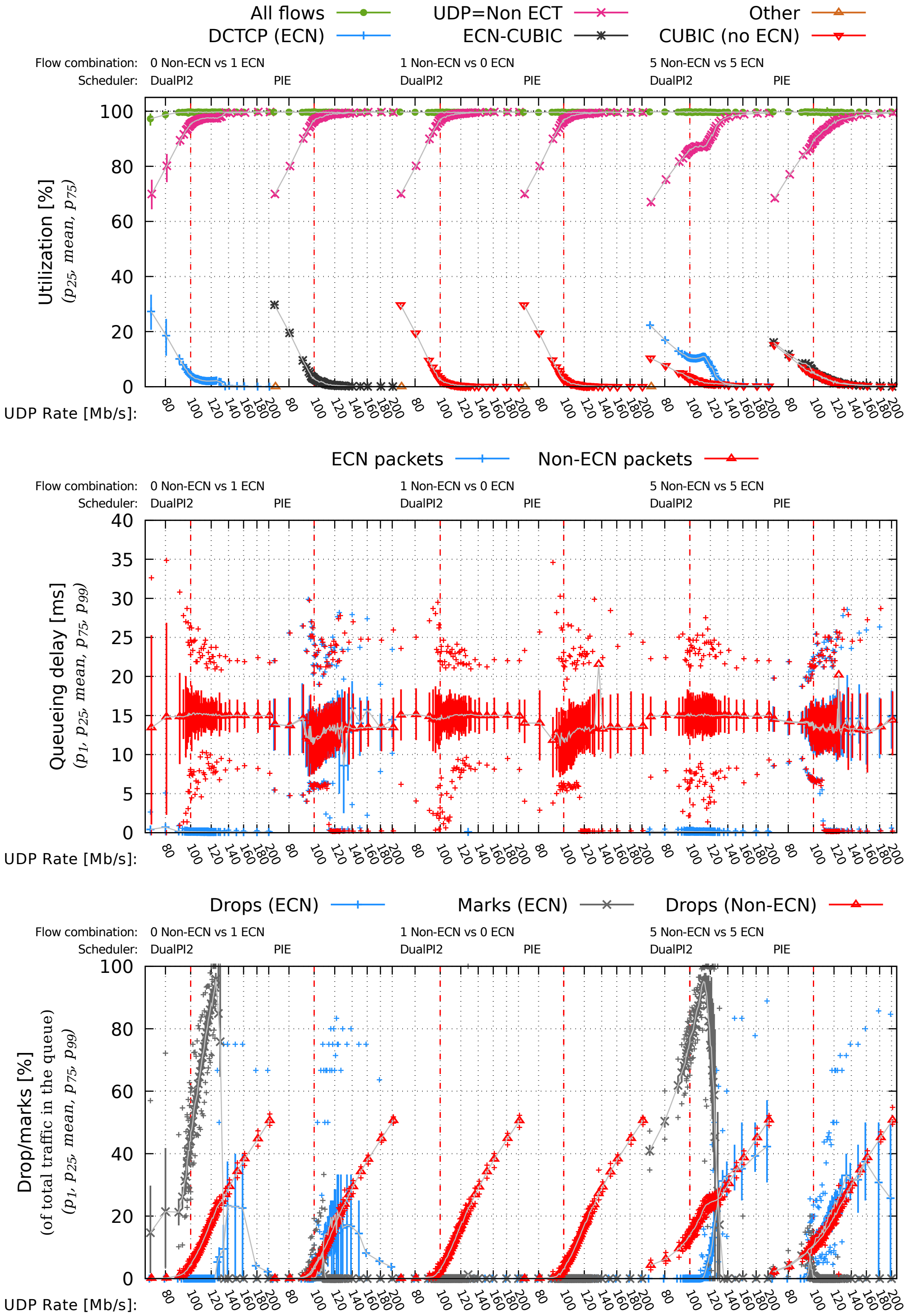


Figure 12.6: Testing overload with existing traffic. Overload is done without ECT, i.e. with the classic (non-ECN) traffic. RTT is 10 ms. Linkrate 100 Mbit/s. The red line represents UDP traffic at link rate.

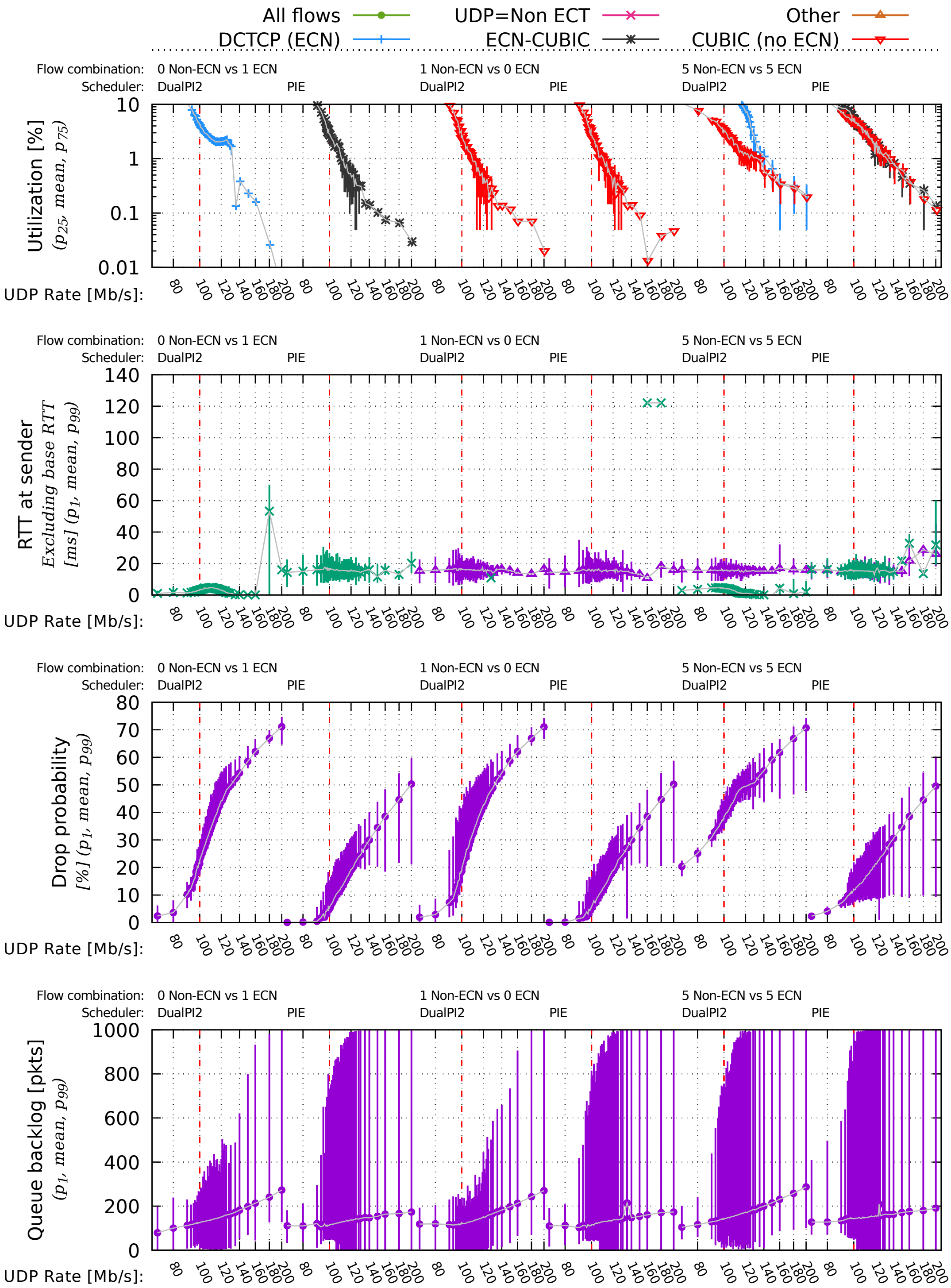


Figure 12.7: Addition to figure 12.6. The first plot shows the utilization again but with only the lower 10 percent in a logarithmic scale.

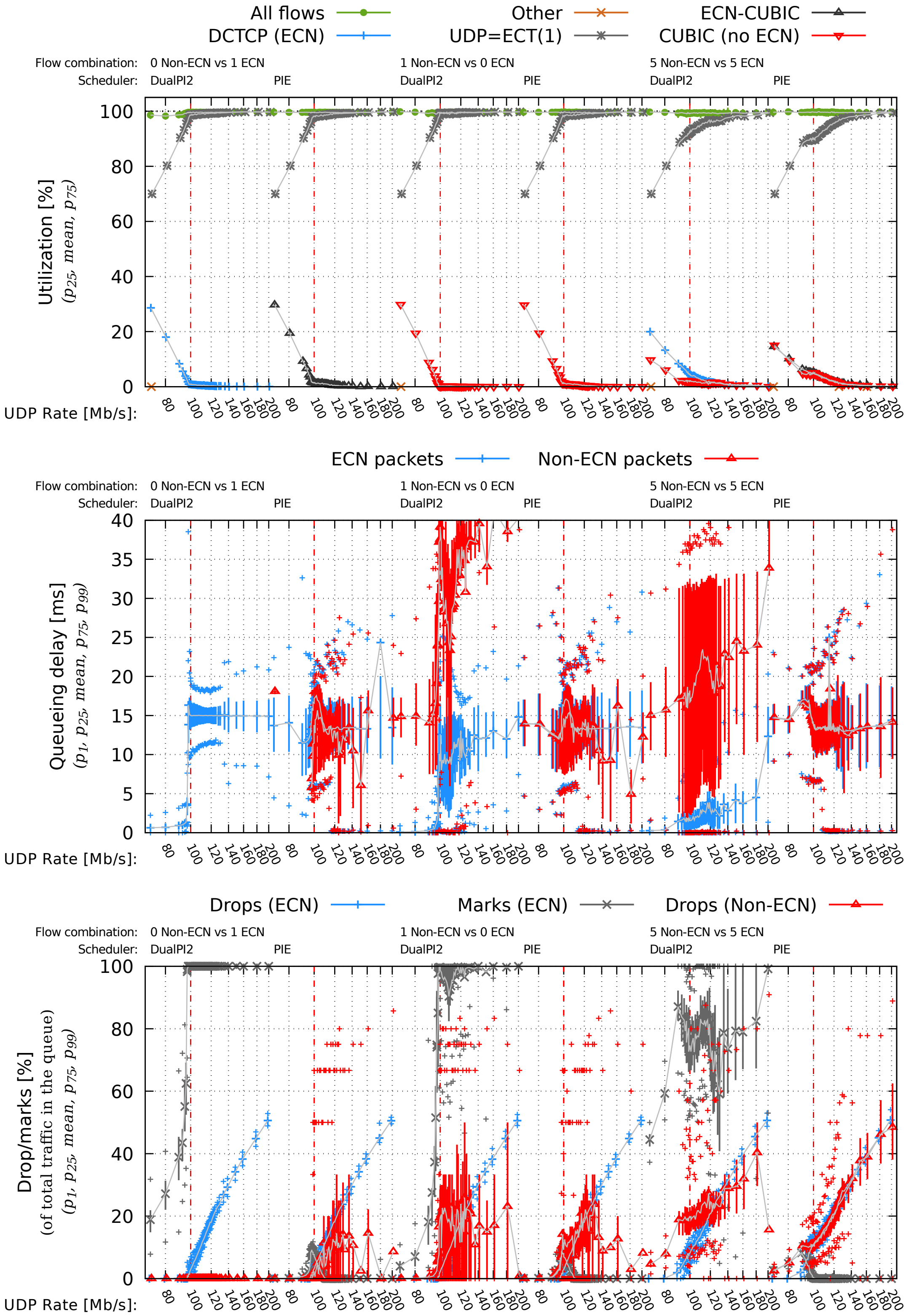


Figure 12.8: Testing overload with existing traffic. Overload is done with ECT(1), i.e. with the scalable (ECN) traffic. RTT is 10 ms. Linkrate 100 Mbit/s. The red line represents UDP traffic at link rate.

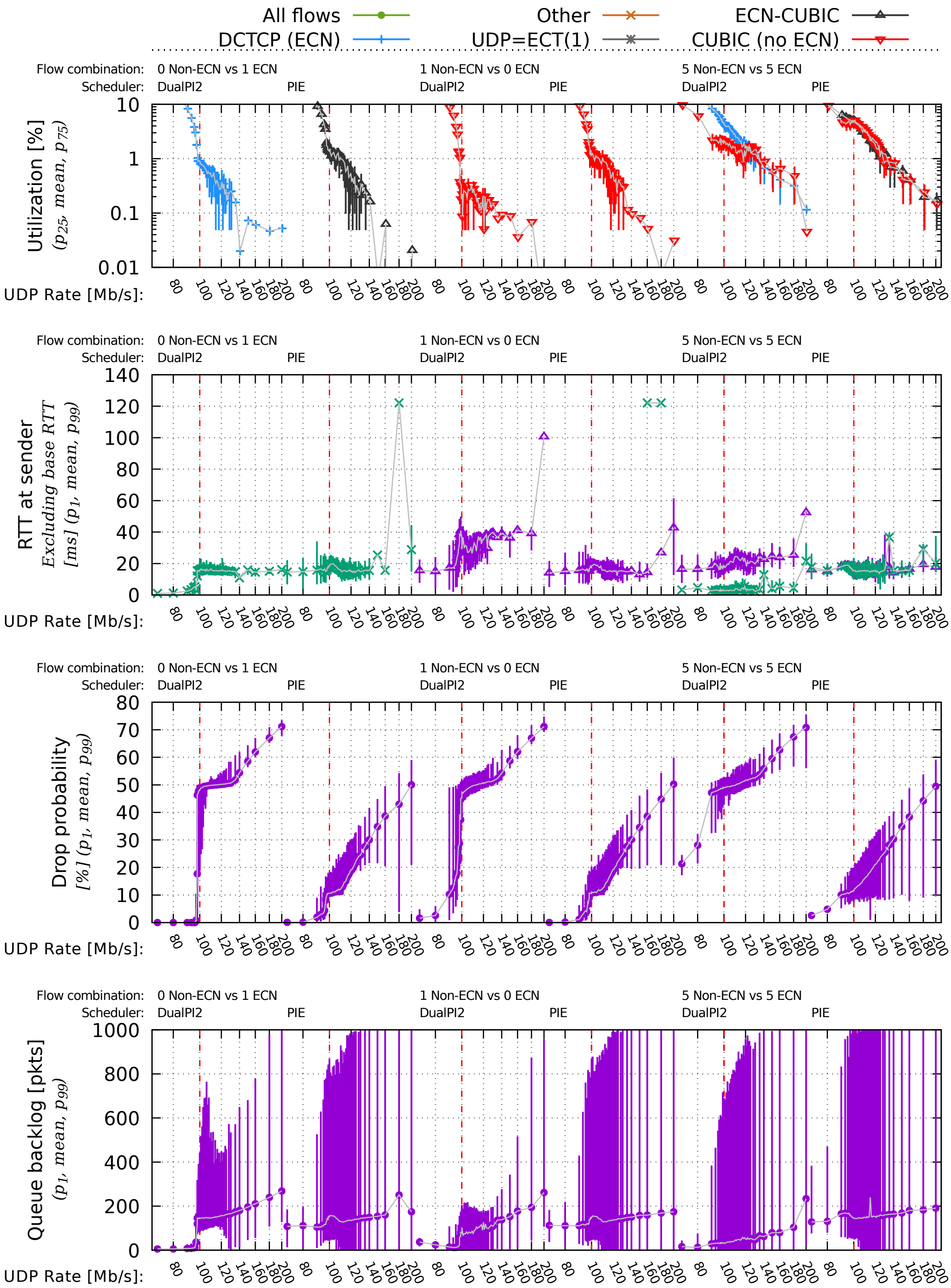


Figure 12.9: Addition to figure 12.8. The first plot shows the utilization again but with only the lower 10 percent in a logarithmic scale.

Chapter 13

Ultra-low queueing delay threshold

DualPI2 uses a shallow threshold for targeting scalable traffic in the L4S queue. Packets are marked as soon as they go above the threshold. The default threshold in the reference implementation and from the DualPI2 paper uses a threshold of 1 ms.

To evaluate how this threshold impact the behaviour, I run a series of tests across different link rates and RTTs to see how stable the connection is and what utilization we can achieve. The test written for this is given in listing 21.

The results are given in figure 13.1. The results clearly shows there are issues with this threshold. At very low BDP the flow achieves near 100 % link utilization, but e.g. at 100 Mbit/s the utilization starts to drop between 2 and 5 ms of RTT. There is also an odd behaviour where the utilization seem to rise after first dropping, and then going down again. The location of this seems not to happen at a fixed RTT, as it happens at around 14 ms RTT at 100 Mbit/s, and at 10 ms RTT at 200 Mbit/s. At 400 Mbit/s the flow seems unstable even at very low RTTs. I have not been able to understand why this drop is happening.

As can be seen from the plot estimating the window size of the flows, the link utilization usualy drops when the window goes above above 12-13 packets, nontheless what the RTT or bitrate is.

One possible reason for this behaviour is that the threshold is based on the idea that all packets are paced perfectly. However, most likely there will be a variation in queueing delay due to scheduling,

[25] gives insight into using an instantaneously queue length for marking ECN packets, and shows that it causes under-utilization, much like what we are experiencing here.

Setting the threshold to 5 ms instead of 1 ms, as shown with the results in figure 13.2, the utilization greatly improves.

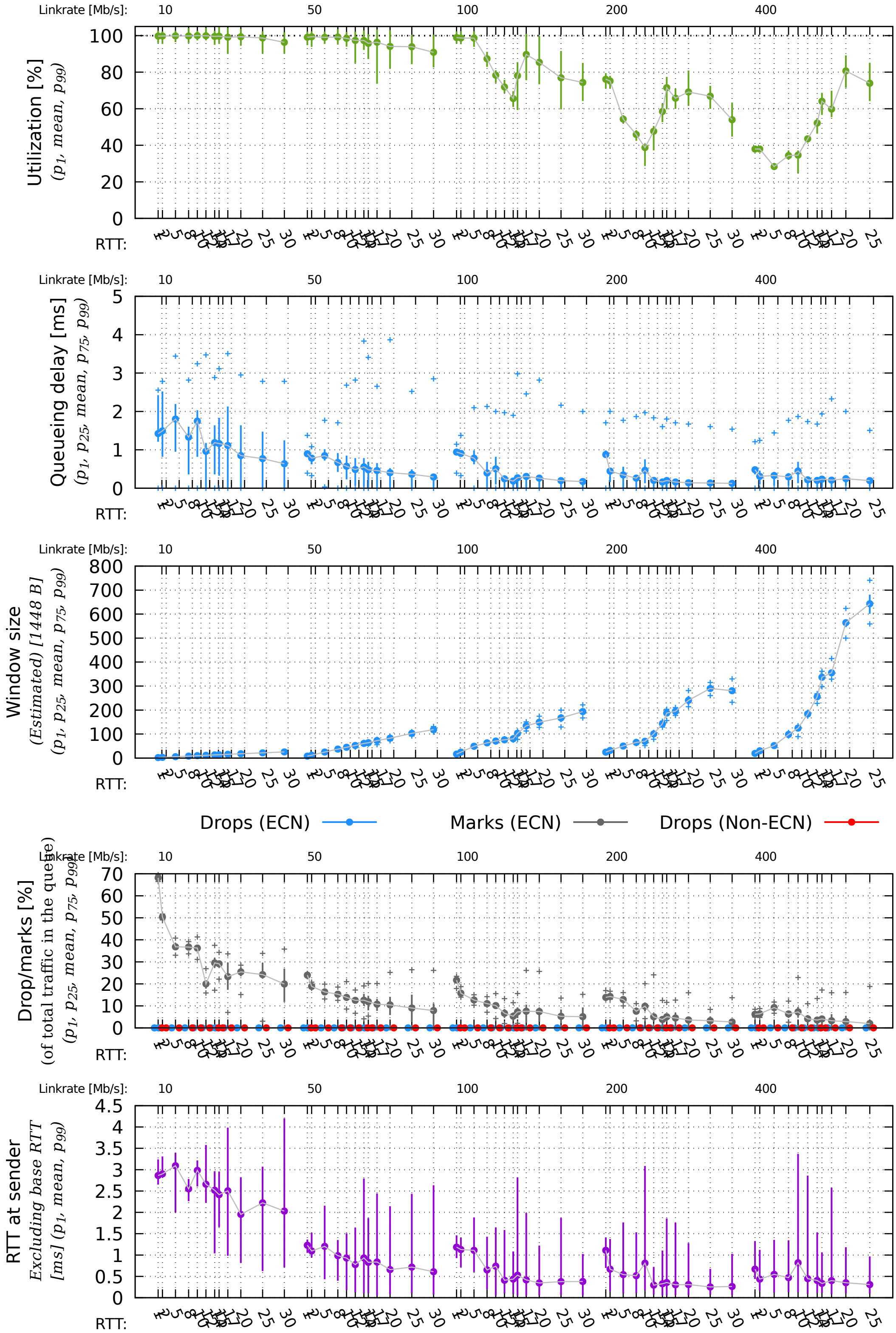


Figure 13.1: Testing threshold for marking of DualPI2. One flow DCTCP. Threshold is set to the default of 1 ms.

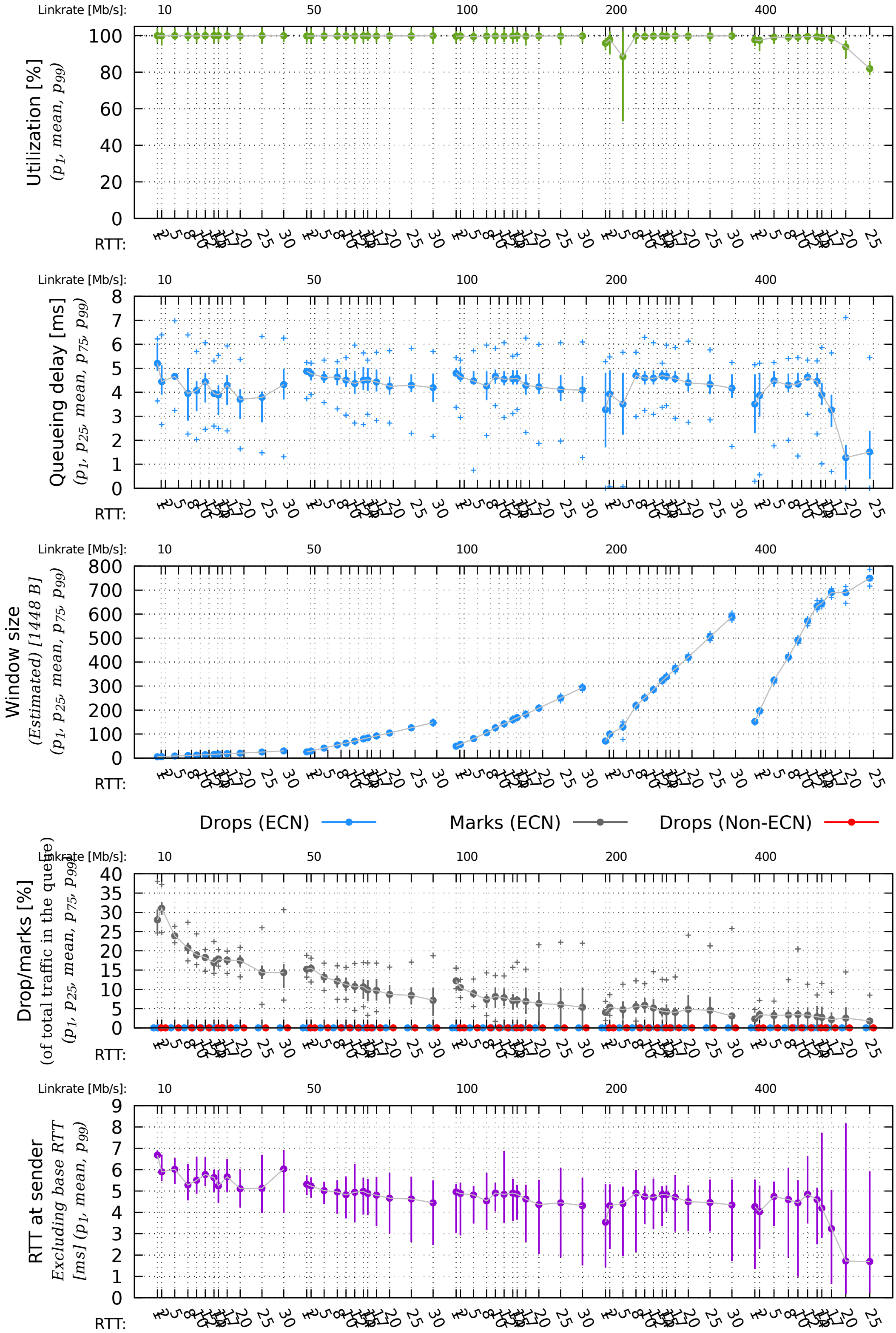


Figure 13.2: Testing threshold for marking of DualPI2. One flow DCTCP. Threshold is set to 5 ms.

Chapter 14

Comparing virtual tests against the physical testbed

The previous results are run in a Docker environment, on top of a mostly idle server. However, running the same test in the physical testbed, we get slightly different results. Figure 14.1 shows results that can be compared about the test in the virtual environment shown in figure 13.1.

Similar, for the overload tests, figure 14.2 gives a comparison against figure 12.8. From the tests of overload it seems the results gives the same understanding of the test.

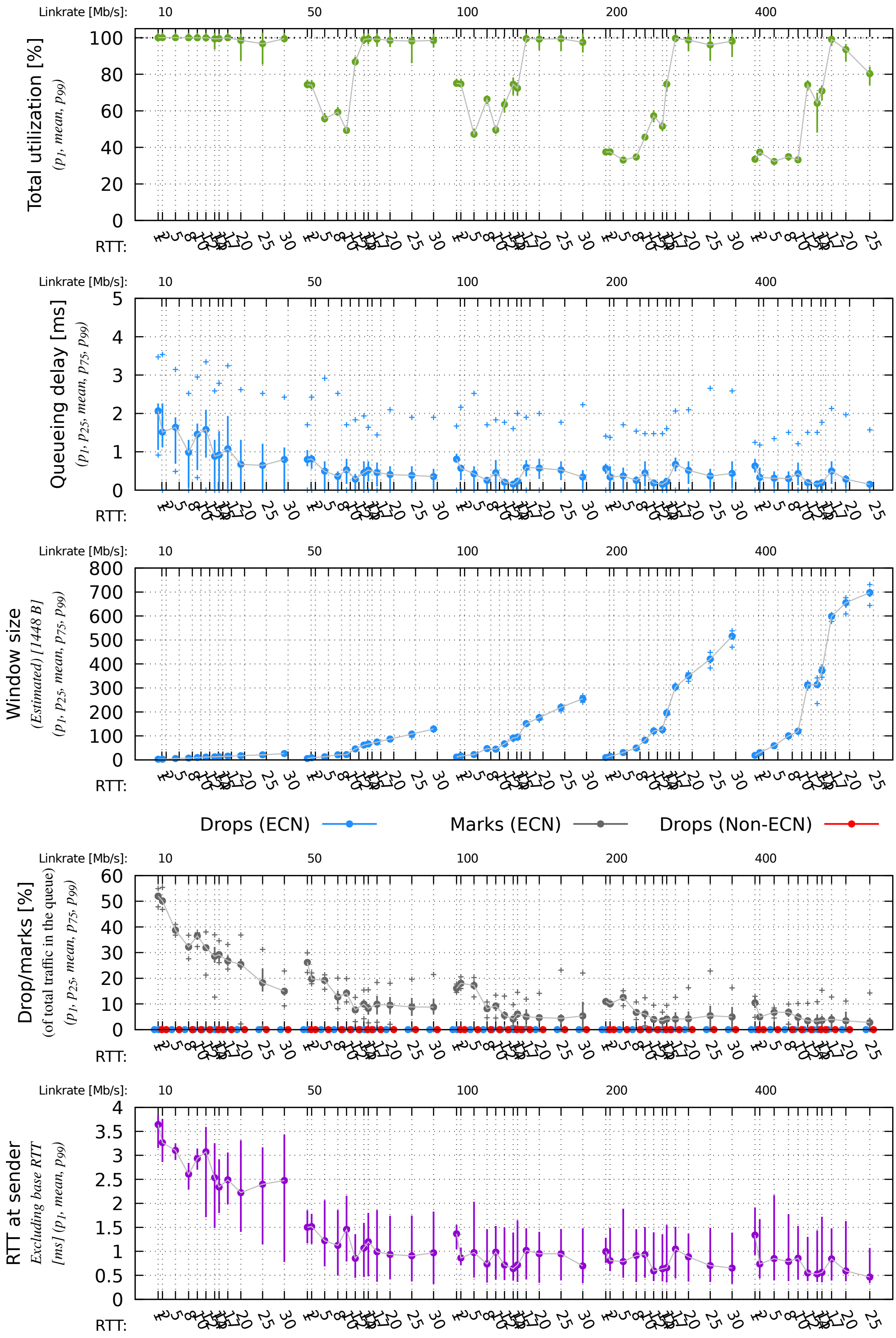


Figure 14.1: Testing threshold for marking of DualPI2. One flow DCTCP. Threshold is set to the default of 1 ms. Run in the physical testbed.

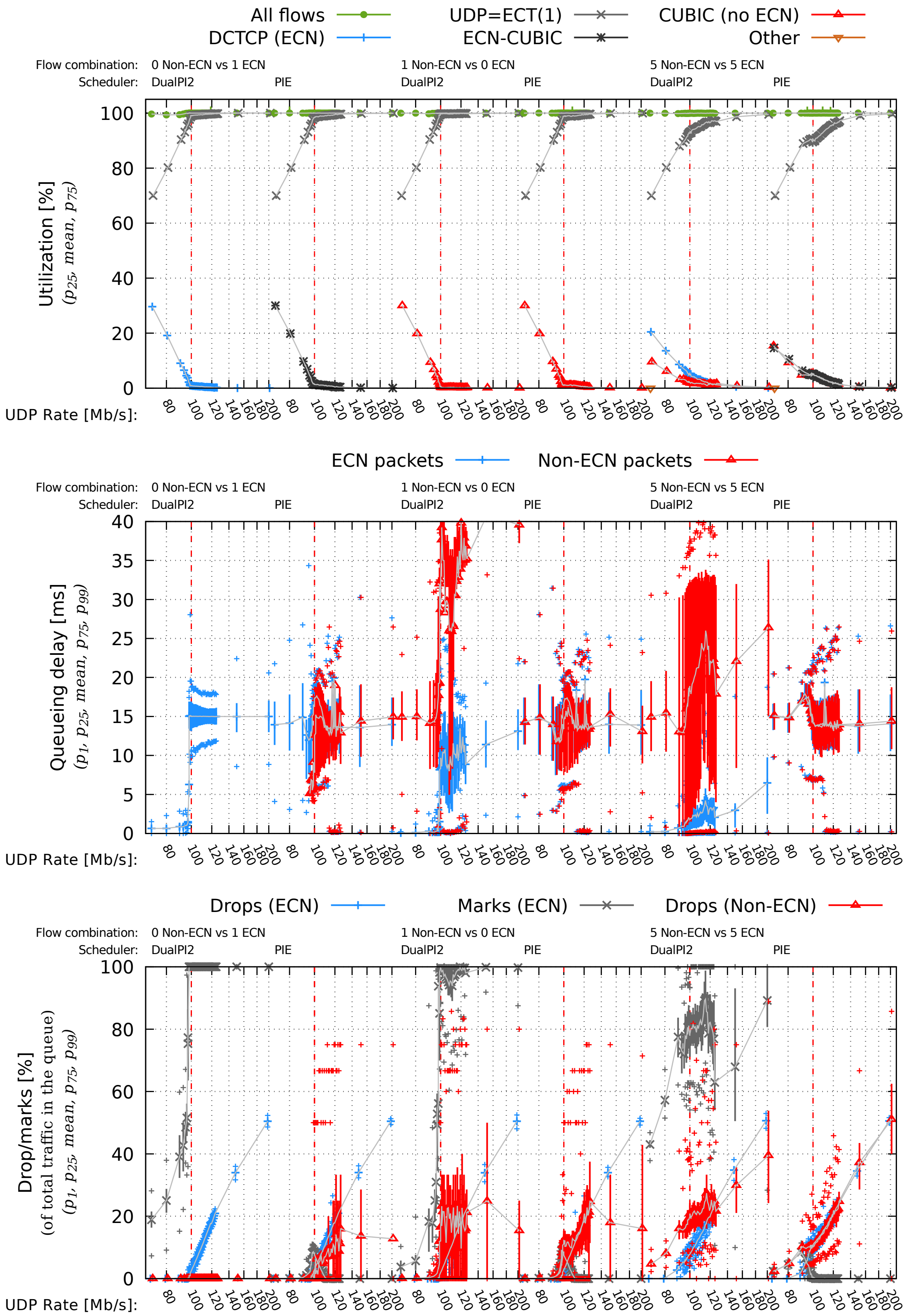


Figure 14.2: Comparison against figure 12.8 which is run in Docker. This figure shows the test run in the physical testbed.

Part IV

Conclusion

Chapter 15

Conclusion

In this thesis I presented my test framework developed for running a large variety of tests against DualPI2. Far from all tests found its way into this thesis, but the tool has proven very useful to evaluate and investigate the different properties various configuration yields. By open sourcing the framework I hope others will find use for it.

I have used the test framework to present overload results of DualPI2 as well as evaluating the queueing threshold of scalable traffic such as DCTCP. The results shows there are some cases DualPI2 have issues, but overall the results looks very promising.

Chapter 16

Future Work

16.1 Testing scenarios

In this thesis I have focused mostly on greedy long-living TCP flows as part of overloading. It would be interesting to also include thin flows and other short living flows such as simulating HTTP traffic. Especially how the AQM performs with varying packet sizes is interesting. This is however left as future work.

Our testbed have been limited to 1 GigE hardware, and as such I have only ran tests with a bitrate lower than this. Testing the different scenarious on 10 GigE hardware is left as future work.

16.2 Easier instrumentation of other AQMs

A concern with the current test framework is that it requires modifications to the existing schedulers. The modifications are also prone to errors if trying to instrument an advanced scheduler. One idea that is left as possible future work is to create a instrumentation qdisc that can be put outside the scheduler/AQM being tested. This way no modifications would be required. This would make it possible to add queueing delay instrumentation. Drop instrumentation however is more difficult, as which packets are being dropped is unknown, and might not be possible to instrument correctly between ECN and non-ECN traffic.

16.3 Malicious users

I have not been investigating how to handle a malicious user. All though overload can be seen similar to a malicious user, such a user is probably able to cause a higher degree of overload and unstability in the system, than what I have investigated. Security concerns such as this has not been investigated.

16.4 Segmentation offloading

All test cases has been run without various segmentation offloading, as described in section 9.1. Enabling such options might cause a change in how the AQM behaves, especially due to the larger packets that will be present in the queue. [25] also shows us that segmentation offloading also causes micro-burst that might further affect the behaviour of DualPI2 and should be investigated.

16.5 Stability and accuracy of using a virtual testbed

The tests that have been presented in this thesis have mostly been run both in the physical testbed and in Docker. I have seen some situations where results might be different, e.g. the one described in chapter 14. I have not extensively evaluated in which conditions it migth fail, and to what extent it can be used for accurate evaluations.

My theory is however that it probably is comparable within the same machine, but that comparing with other testbeds might yield different results. The same uncertainty relies with the physical testbed, as the results might be different between physical testbeds as well.

Bibliography

[1] Richelle Adams. 'Active Queue Management: A Survey'. In: *IEEE Communications Surveys & Tutorials* 15.3 (2013), pp. 1425–1476. DOI: http://dx.doi.org/10.1109/SURV.2012.082212.00018.

[2] Mohammad Alizadeh et al. 'Data center TCP (DCTCP)'. In: *SIGCOMM Comput. Commun. Rev.* 41.4 (Aug. 2010). ISSN: 0146-4833. URL: http://dl.acm.org/citation.cfm?id=2043164.1851192.

[3] Fred Baker and Gorry Fairhurst. *IETF Recommendations Regarding Active Queue Management*. Request for Comments RFC7567. RFC Editor, July 2015. URL: https://tools.ietf.org/html/rfc7567.

[4] David Black. *Explicit Congestion Notification (ECN) Experimentation*. Internet Draft draft-ietf-tsvwg-ecn-experimentation-00. (Work in Progress). Internet Engineering Task Force, Dec. 2016. URL: http://tools.ietf.org/html/draft-black-tsvwg-ecn-experimentation.

[5] B. Braden et al. *Recommendations on Queue Management and Congestion Avoidance in the Internet*. Request for Comments 2309. RFC Editor, Apr. 1998. URL: http://www.ietf.org/rfc/rfc2309.txt.

[6] Koen De Schepper et al. ''Data Centre to the Home': Deployable Ultra-Low Queuing Delay for All'. In: (Under Submission). Jan. 2017.

[7] Koen De Schepper et al. 'PI^2 : A Linearized AQM for both Classic and Scalable TCP'. In: *Proc. ACM CoNEXT 2016*. New York, NY, USA: ACM, Dec. 2016, pp. 105–119. ISBN: 978-1-4503-4297-1.

[8] S. Deering and R. Hinden. *Internet Protocol, Version 6 (IPv6) Specification*. Request for Comments 2460. RFC Editor, Dec. 1998. URL: http://www.ietf.org/rfc/rfc2460.txt.

[9] *Ethernet Address Resolution Protocol: Or Converting Network Protocol Addresses to 48.bit Ethernet Address for Transmission on Ethernet Hardware*. RFC 826. Nov. 1982. DOI: 10.17487/rfc826. URL: https://rfc-editor.org/rfc/rfc826.txt.

[10] Sally Floyd, Ramakrishna Gummadi and Scott Shenker. 'Adaptive RED: An Algorithm for Increasing the Robustness of RED's Active Queue Management'. http://www.icir.org/floyd/papers/adaptiveRed.pdf. Aug. 2001.

[11] Sangtae Ha, Injong Rhee and Lisong Xu. 'CUBIC: a new TCP-friendly high-speed TCP variant'. In: *SIGOPS Operating Systems Review* 42.5 (July 2008), pp. 64–74. ISSN: 0163-5980. DOI: 10.1145/1400097.1400105. URL: http://doi.acm.org/10.1145/1400097.1400105.

[12] B. Hamzeh et al. 'DOCSIS 3.1: scaling broadband cable to Gigabit speeds'. In: *IEEE Communications Magazine* 53.3 (Mar. 2015), pp. 108–113. ISSN: 0163-6804. DOI: 10.1109/MCOM.2015.7060490.

[13] Tom Henderson and Sally Floyd. *The NewReno Modification to TCP's Fast Recovery Algorithm*. RFC 2582. Apr. 1999. DOI: 10.17487/rfc2582. URL: https://rfc-editor.org/rfc/rfc2582.txt.

[14] C. V. Hollot et al. 'On Designing Improved Controllers for AQM Routers Supporting TCP Flows'. In: *Proc. INFOCOM 2001. 20th Annual Joint Conf. of the IEEE Computer and Communications Societies.* 2001, pp. 1726–1734. URL: http://www.ieee-infocom.org/2001/paper/792.pdf.

[15] Van Jacobson. 'Congestion Avoidance and Control'. In: *Proc. ACM SIGCOMM'88 Symposium, Computer Communication Review* 18.4 (Aug. 1988), pp. 314–329. URL: http://citeseer.ist.psu.edu/jacobson88congestion.html.

[16] J. Nagle. *Congestion control in IP / TCP internetworks*. Request for Comments 896. (Status: unknown). RFC Editor, Jan. 1984. URL: http://www.ietf.org/rfc/rfc896.txt.

[17] Linux Kernel Newbies. *Linux 2.6.19*. Nov. 2006. URL: https://kernelnewbies.org/Linux_2_6_19.

[18] Rong Pan et al. 'PIE: A Lightweight Control Scheme To Address the Bufferbloat Problem'. In: *High Performance Switching and Routing (HPSR'13)*. IEEE. 2013.

[19] Heather J. Patrick, Alan D. Kersey and Frank Bucholtz. 'Analysis of the Response of Long Period Fiber Gratings to External Index of Refraction'. In: *J. Lightwave Technol.* 16.9 (Sept. 1998), p. 1606. URL: http://jlt.osa.org/abstract.cfm?URI=jlt-16-9-1606.

[20] Jon Postel (Ed.) *Internet Protocol*. STD 5. RFC 791. RFC Editor, Sept. 1981. URL: http://www.ietf.org/rfc/rfc791.txt.

[21] Jon Postel (Ed.) *Transmission Control Protocol*. STD 7. RFC 793. RFC Editor, Sept. 1981. URL: http://www.ietf.org/rfc/rfc793.txt.

[22] Jon Postel. *User Datagram Protocol*. STD 6. RFC 768. RFC Editor, Aug. 1980. URL: http://www.ietf.org/rfc/rfc768.txt.

[23] K. K. Ramakrishnan, Sally Floyd and David Black. *The Addition of Explicit Congestion Notification (ECN) to IP*. Request for Comments 3168. RFC Editor, Sept. 2001. URL: http://www.ietf.org/rfc/rfc3168.txt.

[24] Chris Rapier and Benjamin Bennett. *Enabling High performance Bulk Data Transfers With SSH*. 2008. URL: https://www.slideshare.net/datacenters/enabling-high-performance-bulk-data-transfers-with-ssh.

[25] Danfeng Shan and Fengyuan Ren. 'Improving ECN Marking Scheme with Micro-burst Traffic in Data Center Networks'. In: (May 2017).

[26] Verizon. *IP Latency Statistics*. URL: http://www.verizonenterprise.com/about/network/latency/.

[27] *What is Docker*. 2017. URL: https://www.docker.com/what-docker.

[28] Greg White and Rong Pan. *Active Queue Management (AQM) Based on Proportional Integral Controller Enhanced PIE) for Data-Over-Cable Service Interface Specifications (DOCSIS) Cable Modems*. RFC 8034. Feb. 2017. DOI: 10.17487/rfc8034. URL: https://rfc-editor.org/rfc/rfc8034.txt.

[29] P. Yang et al. 'TCP Congestion Avoidance Algorithm Identification'. In: *IEEE/ACM Transactions on Networking* 22.4 (Aug. 2014), pp. 1311–1324. ISSN: 1063-6692. DOI: 10.1109/TNET.2013.2278271.

Appendices

Appendix A

Source code

A.1 Greedy

See also https://github.com/henrist/greedy for the complete release.

```
#include <errno.h>
#include <linux/tcp.h>
#include <netdb.h>
#include <netinet/in.h>
#include <pthread.h>
#include <signal.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <strings.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <sys/utsname.h>
#include <time.h>
#include <unistd.h>

#define DEFAULT_REPORT_MS 250
#define DEFAULT_BUF_SIZE 524288

char *buffer;
int buffer_size = DEFAULT_BUF_SIZE;
int exit_program = 0;
char *hostname;
int keep_running = 0;
int listen_sockfd = -1;
int log_running = 0;
pthread_t log_thread;
enum { MODE_CLIENT, MODE_SERVER } mode = MODE_CLIENT;
int nonblock = 1;
int portno;
int report_ms = DEFAULT_REPORT_MS;
int syscall_started;
int syscall_finished;
int tcp_notsent_capability = 0;
long long total_bytes;
long long total_bytes_buf;
```

```c
int verbose = 0;

struct bytes_report {
    float val;
    char suffix[4];
    char repr[50];
};

void logging_thread_run(void *arg);

void int_handler(int dummy) {
    exit_program = 1;

    if (listen_sockfd != -1) {
        close(listen_sockfd);
        listen_sockfd = -1;
    }
}

void print_usage(char *argv[]) {
    fprintf(stderr,
        "Usage client: %s <host> <port>\n"
        "Usage server: %s -s <port>\n"
        "Options:\n"
        "  -b n  buffer size in bytes to read/write call (default: %d)\n"
        "  -r    keep server running when client disconnect\n"
        "  -t n  report every n milliseconds, implies -vv (default: %d)\n"
        "  -v    verbose output (more verbose if multiple -v)\n"
        "  -w    block on tcp send\n",
        argv[0],
        argv[0],
        DEFAULT_BUF_SIZE,
        DEFAULT_REPORT_MS);
}

void parse_arg(int argc, char *argv[]) {
    int opt;

    while ((opt = getopt(argc, argv, "b:rst:vw")) != -1) {
        switch (opt) {
            case 'b':
                buffer_size = atoi(optarg);
                break;
            case 'r':
                keep_running = 1;
                break;
            case 's':
                mode = MODE_SERVER;
                break;
            case 't':
                report_ms = atoi(optarg);
                if (verbose < 2) {
                    verbose = 2;
                }
                break;
            case 'v':
                verbose += 1;
```

```
                break;
            case 'w':
                nonblock = 0;
                break;
            default:
                print_usage(argv);
                exit(1);
        }
    }

    if (argc - optind < (mode == MODE_SERVER ? 1 : 2)) {
        print_usage(argv);
        exit(1);
    }

    if (mode == MODE_SERVER) {
        portno = atoi(argv[optind]);
    } else {
        hostname = malloc(strlen(argv[optind]));
        memcpy(hostname, argv[optind], strlen(argv[optind]));

        portno = atoi(argv[optind+1]);
    }
}

void start_logger(int sockfd) {
    int pret;
    pret = pthread_create(&log_thread, NULL, (void *) &logging_thread_run, (void *) (intptr_t) sockfd);
    if (pret != 0) {
        fprintf(stderr, "Could not create logging thread\n");
    } else {
        log_running = 1;
    }
}

void stop_logger() {
    if (log_running) {
        pthread_cancel(log_thread);
    }
}

void set_tcp_nodelay(int sockfd) {
    int enable = 1;
    if (setsockopt(sockfd, IPPROTO_TCP, TCP_NODELAY, (void *) &enable, sizeof(enable)) < 0) {
        fprintf(stderr, "setsockopt(TCP_NODELAY) failed");
        exit(1);
    }
}

void get_bytes_format(long long value, struct bytes_report *br, int align) {
    char fmt[20];
    br->val = value;

    if (br->val > 1024) {
        if (br->val > 1024) {
            br->val /= 1024;
```

```
            strcpy(br->suffix, "KiB");
        }

        if (br->val > 1024) {
            br->val /= 1024;
            strcpy(br->suffix, "MiB");
        }

        if (br->val > 1024) {
            br->val /= 1024;
            strcpy(br->suffix, "GiB");
        }

        if (align > 0) {
            sprintf(fmt, "%%%d.3f %%s", align-4);
        } else {
            sprintf(fmt, "%%.3f %%s");
        }

        sprintf(br->repr, fmt, br->val, br->suffix);
    }

    else {
        strcpy(br->suffix, "B");

        if (align > 0) {
            sprintf(fmt, "%%%d.0f       %%s", align-8);
        } else {
            sprintf(fmt, "%%.0f %%s");
        }

        sprintf(br->repr, fmt, br->val, br->suffix);
    }
}

void report_closed() {
    if (total_bytes > 0) {
        struct bytes_report br;
        get_bytes_format(total_bytes, &br, 0);

        printf("finished, a total number of %s was %s, %.2f %% of %s buffer used\n",
            br.repr,
            mode == MODE_SERVER ? "written" : "read",
            (float) total_bytes / (float) total_bytes_buf * 100,
            mode == MODE_SERVER ? "write" : "read");
    }
}

void run_client() {
    int read_bytes;
    struct sockaddr_in serv_addr;
    struct hostent *server;
    int sockfd;

    server = gethostbyname(hostname);
    if (server == NULL) {
        fprintf(stderr, "No such host %s\n", hostname);
        exit(1);
```

```
    }

    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd < 0) {
        fprintf(stderr, "Error opening socket\n");
        exit(1);
    }

    set_tcp_nodelay(sockfd);

    bzero((char *) &serv_addr, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    bcopy((char *) server->h_addr, (char *) &serv_addr.sin_addr.s_addr, server->h_length);
    serv_addr.sin_port = htons(portno);

    if (connect(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0) {
        fprintf(stderr, "Error connecting to server\n");
        exit(1);
    }

    if (verbose >= 2) {
        start_logger(sockfd);
    }

    syscall_started = 0;
    syscall_finished = 0;
    total_bytes = 0;
    total_bytes_buf = 0;

    //bzero(buffer, buffer_size);
    do {
        syscall_started++;
        read_bytes = read(sockfd, buffer, buffer_size);
        syscall_finished++;

        if (read_bytes > 0) {
            if (verbose >= 4) {
                printf(".");
            }
            total_bytes += read_bytes;
            total_bytes_buf += buffer_size;
        } else if (verbose >= 3) {
            printf("  read=0  ");
        }
    } while (read_bytes > 0 && !exit_program);

    if (verbose) {
        report_closed();
    }

    close(sockfd);
}

void run_server() {
    struct sockaddr_in cli_addr;
    int clilen;
    struct sockaddr_in serv_addr;
```

```c
    int sockfd;
    int wrote_bytes;

    listen_sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (listen_sockfd < 0) {
        fprintf(stderr, "Error opening socket\n");
        exit(1);
    }

    int enable = 1;
    if (setsockopt(listen_sockfd, SOL_SOCKET, SO_REUSEADDR, &enable, sizeof(enable)) < 0) {
        fprintf(stderr, "setsockopt(SO_REUSEADDR) failed");
        exit(1);
    }

    bzero((char *) &serv_addr, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = INADDR_ANY;
    serv_addr.sin_port = htons(portno);

    if (bind(listen_sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0) {
        fprintf(stderr, "Error binding socket\n");
        exit(1);
    }

    listen(listen_sockfd, 5);
    clilen = sizeof(cli_addr);

    do {
        if (verbose) {
            printf("waiting for client to connect\n");
        }

        sockfd = accept(listen_sockfd, (struct sockaddr *) &cli_addr, &clilen);
        if (exit_program) {
            return;
        }
        if (sockfd < 0) {
            fprintf(stderr, "Error accepting socket\n");
            exit(1);
        }

        set_tcp_nodelay(sockfd);

        if (verbose >= 2) {
            start_logger(sockfd);
        }

        syscall_started = 0;
        syscall_finished = 0;
        total_bytes = 0;
        total_bytes_buf = 0;

        struct timespec sleeptime;
        sleeptime.tv_sec = 0;

```

```
        bzero(buffer, buffer_size);
        int zerosends = 0;
        int backoff;
        while (!exit_program) {
            syscall_started++;
            wrote_bytes = send(sockfd, buffer, buffer_size, nonblock ? MSG_DONTWAIT : 0);
            syscall_finished++;

            if (wrote_bytes == 0) {
                fprintf(stderr, "unexpected send of 0 bytes\n");
                break;
            } else if (wrote_bytes < 0) {
                if (errno == EAGAIN || errno == EWOULDBLOCK) {
                    zerosends++;

                    backoff = zerosends * 10; // base of 10 ms
                    if (backoff >= 1000) backoff = 999;
                    sleeptime.tv_nsec = backoff * 1000000;

                    if (verbose >= 4) {
                        printf("  send=0, backoff=%d  ", backoff);
                    }

                    nanosleep(&sleeptime, NULL);
                    continue;
                } else {
                    fprintf(stderr, "send failed with errno: %d\n", errno);
                    break;
                }
            }

            zerosends = 0;
            total_bytes += wrote_bytes;
            total_bytes_buf += buffer_size;

            if (verbose >= 4) {
                printf(".");
            }
        }

        if (verbose) {
            report_closed();
        }

        stop_logger();
        close(sockfd);
    } while (keep_running && !exit_program);

    if (listen_sockfd != -1) {
        close(listen_sockfd);
    }
}

void detect_tcp_notsent_capability() {
    struct utsname unamedata;
    int v1, v2;

```

```c
    // tcp_info.tcpi_notsent_bytes is available since Linux 4.6
    if (uname(&unamedata) == 0 && sscanf(unamedata.release, "%d.%d.", &v1, &v2) == 2) {
        if (v1 > 4 || (v1 == 4 && v2 >= 6)) {
            tcp_notsent_capability = 1;
        }
    }
}

int main(int argc, char *argv[])
{
    detect_tcp_notsent_capability();
    signal(SIGINT, int_handler);
    signal(SIGPIPE, SIG_IGN);
    parse_arg(argc, argv);

    buffer = malloc(buffer_size);
    if (buffer == NULL) {
        fprintf(stderr, "Could not allocate memory for buffer (%d bytes)\n", buffer_size);
        exit(1);
    }

    if (mode == MODE_SERVER) {
        run_server();
    } else {
        run_client();
    }

    free(buffer);
    return 0;
}

void logging_thread_run(void *arg)
{
    int sockfd = (intptr_t) arg;
    long long prev_total_bytes = 0;
    int prev_syscall_finished = 0;
    long long cur_total_bytes;
    int cur_syscall_finished;
    struct timespec sleeptime;
    struct bytes_report br;
    int s_rcv, s_snd, len;

    sleeptime.tv_sec = report_ms / 1000;
    sleeptime.tv_nsec = (report_ms % 1000) * 1000000;

    printf("stats: reports every %d ms, sb = SO_SNDBUF, rb = SO_RCVBUF\n", report_ms);
    printf("R = RTT, F = packets in flight, L = loss, W = window size\n");

    while (1) {
        struct tcp_info info;
        len = sizeof(struct tcp_info);
        if (getsockopt(sockfd, IPPROTO_TCP, TCP_INFO, &info, &len) != 0) {
            fprintf(stderr, "getsockopt(TCP_INFO) failed, errno: %d\n", errno);
```

```
            break;
        }

        int in_flight = info.tcpi_unacked - (info.tcpi_sacked + info.tcpi_lost) + info.tcpi_retrans;

        int syscall_in_progress = syscall_finished != syscall_started;
        cur_total_bytes = total_bytes;
        cur_syscall_finished = syscall_finished;

        get_bytes_format(cur_total_bytes - prev_total_bytes, &br, 12);

        len = sizeof(s_rcv);
        if (getsockopt(sockfd, SOL_SOCKET, SO_RCVBUF, &s_rcv, &len) < 0) {
            fprintf(stderr, "getsockopt(SO_RCVBUF) failed");
            break;
        }

        len = sizeof(s_snd);
        if (getsockopt(sockfd, SOL_SOCKET, SO_SNDBUF, &s_snd, &len) < 0) {
            fprintf(stderr, "getsockopt(SO_SNDBUF) failed");
            break;
        }

        printf("%4d%s %s",
            cur_syscall_finished - prev_syscall_finished,
            mode == MODE_SERVER
                ? (syscall_in_progress ? "W" : "w")
                : (syscall_in_progress ? "R" : "r"),
            br.repr);

        printf(" R=%7.2f/%5.2f F=%5d",
            (double) info.tcpi_rtt/1000,
            (double) info.tcpi_rttvar/1000,
            in_flight);

        if (info.tcpi_lost == 0) {
            printf(" L=%5s", "-");
        } else {
            printf(" L=%5u", info.tcpi_lost);
        }

        printf(" rto=%7.2f", (double) info.tcpi_rto / 1000);

        printf(" W=%5d retrans=%3u/%u",
            info.tcpi_snd_cwnd,
            info.tcpi_retrans,
            info.tcpi_total_retrans);

        if (tcp_notsent_capability) {
            printf(" notsent=%7d b", info.tcpi_notsent_bytes);
        }

        if (info.tcpi_options & TCPI_OPT_ECN)
            printf(" ecn");
```

```
        if (info.tcpi_options & TCPI_OPT_ECN_SEEN)
            printf("S");

        printf(" rb=%d sb=%d", s_rcv, s_snd);

        printf("\n");

        prev_total_bytes = cur_total_bytes;
        prev_syscall_finished = cur_syscall_finished;
        nanosleep(&sleeptime, NULL);
    }
}
```

Listing 8: A simple client/server which attempts to always have data in the Linux TCP stack available to dequeue to the network. It basicly tries to fill the TCP window at all times.

A.2 Testbed setup

```
configure_host_cc() {(set -e
    local host=$1
    local tcp_congestion_control=$2
    local tcp_ecn=$3

    local feature_ecn=""
    if [ "$tcp_ecn" == "1" ]; then
        feature_ecn=" features ecn"
    fi

    # the 10.25. range belongs to the Docker setup
    # it needs to use congctl for a per route configuration
    # (congctl added in iproute2 v4.0.0)
    ssh root@$host '
        set -e
        if [ -f /proc/sys/net/ipv4/tcp_congestion_control ]; then
            sysctl -q -w net.ipv4.tcp_congestion_control='$tcp_congestion_control'
        else
            # we are on docker
            . /aqmt-vars-local.sh
            if ip a show $IFACE_AQM | grep -q 10.25.1.; then
                # on client
                ip route replace 10.25.2.0/24 via 10.25.1.2 dev $IFACE_AQM congctl '$tcp_congestion_control$feature_ecn'
                ip route replace 10.25.3.0/24 via 10.25.1.2 dev $IFACE_AQM congctl '$tcp_congestion_control$feature_ecn'
            else
                # on server
                ip_prefix=$(ip a show $IFACE_AQM | grep "inet 10" | awk "{print \$2}" | sed "s/\.[0-9]\+\/.*//")
                ip route replace 10.25.1.0/24 via ${ip_prefix}.2 dev $IFACE_AQM congctl '$tcp_congestion_control$feature_ecn'
            fi
        fi
```

```
        sysctl -q -w net.ipv4.tcp_ecn='$tcp_ecn
) || (echo -e "\nERROR: Failed setting cc $2 (ecn = $3) on node $1\n"; exit 1))

configure_clients_edge_aqm_node() {(set -e
    local testrate=$1
    local rtt=$2
    local aqm_name=$3
    local aqm_params=$4
    local netem_params=$5  # optional

    local delay=$(echo "scale=2; $rtt / 2" | bc)  # delay is half the rtt

    # htb = hierarchy token bucket - used to limit bandwidth
    # netem = used to simulate delay (link distance)

    if [ $rtt -gt 0 ]; then
        if tc qdisc show dev $IFACE_CLIENTS | grep -q "qdisc netem 2:"; then
            tc qdisc change dev $IFACE_CLIENTS handle 2: netem delay ${delay}ms $netem_params
            tc class change dev $IFACE_CLIENTS parent 3: classid 10 htb rate $testrate
        else
            tc qdisc  del dev $IFACE_CLIENTS root 2>/dev/null || true
            tc qdisc  add dev $IFACE_CLIENTS root       handle  1: prio bands 2 priomap 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
            tc filter add dev $IFACE_CLIENTS parent 1:0 protocol ip prio 1 u32 match ip src $IP_AQM_C flowid 1:1
            tc qdisc  add dev $IFACE_CLIENTS parent 1:2 handle 2: netem delay ${delay}ms $netem_params
            tc qdisc  add dev $IFACE_CLIENTS parent 2: handle 3: htb default 10
            tc class  add dev $IFACE_CLIENTS parent 3: classid 10 htb rate $testrate   #burst 1516
        fi
    else
        if ! tc qdisc show dev $IFACE_CLIENTS | grep -q "qdisc netem 2:" && \
            tc qdisc show dev $IFACE_CLIENTS | grep -q "qdisc htb 3:"; then
            tc class change dev $IFACE_CLIENTS parent 3: classid 10 htb rate $testrate
        else
            tc qdisc  del dev $IFACE_CLIENTS root 2>/dev/null || true
            tc qdisc  add dev $IFACE_CLIENTS root       handle  1: prio bands 2 priomap 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
            tc filter add dev $IFACE_CLIENTS parent 1:0 protocol ip prio 1 u32 match ip src $IP_AQM_C flowid 1:1
            tc qdisc  add dev $IFACE_CLIENTS parent 1:2 handle 3: htb default 10
            tc class  add dev $IFACE_CLIENTS parent 3: classid 10 htb rate $testrate   #burst 1516
        fi
```

```
    fi

    if [ -n "$aqm_name" ]; then
        # update params if possible
        if tc qdisc show dev $IFACE_CLIENTS | grep -q "qdisc $aqm_name 15:"; then
            tc qdisc change dev $IFACE_CLIENTS handle 15: $aqm_name $aqm_params
            echo "Updated params on existing aqm"
        else
            tc qdisc  add dev $IFACE_CLIENTS parent 3:10 handle 15: $aqm_name $aqm_params
        fi
    fi
) || (echo -e "\nERROR: Failed configuring AQM clients edge (aqm = $3)\n"; exit 1))

configure_clients_node() {(set -e
    local rtt=$1
    local netem_params=$2  # optional

    local delay=$(echo "scale=2; $rtt / 2" | bc)  # delay is half the rtt

    # netem = used to simulate delay (link distance)

    if [ $rtt -gt 0 ]; then
        hosts=($IP_CLIENTA_MGMT $IP_CLIENTB_MGMT)
        ifaces=($IFACE_ON_CLIENTA $IFACE_ON_CLIENTB)
        for i in ${!hosts[@]}; do
            ssh root@${hosts[$i]} "
                set -e
                # if possible update the delay rather than destroying the existing qdisc
                if tc qdisc show dev ${ifaces[$i]} | grep -q 'qdisc netem 12:'; then
                    tc qdisc change dev ${ifaces[$i]} handle 12: netem delay ${delay}ms $netem_params
                else
                    tc qdisc  del dev ${ifaces[$i]} root 2>/dev/null || true
                    tc qdisc  add dev ${ifaces[$i]} root handle  1: prio bands 2 priomap 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
                    tc qdisc  add dev ${ifaces[$i]} parent 1:2 handle 12: netem delay ${delay}ms $netem_params
                    tc filter add dev ${ifaces[$i]} parent 1:0 protocol ip prio 1 u32 match ip dst $IP_AQM_C flowid 1:1
                fi"
        done
    else
        # no delay: force pfifo_fast
        hosts=($IP_CLIENTA_MGMT $IP_CLIENTB_MGMT)
        ifaces=($IFACE_ON_CLIENTA $IFACE_ON_CLIENTB)
        for i in ${!hosts[@]}; do
            ssh root@${hosts[$i]} "
                set -e
                # skip if already set up
```

```
                if ! tc qdisc show dev ${ifaces[$i]} | grep -q 'qdisc pfifo_fast 1:'; then
                    tc qdisc del dev ${ifaces[$i]} root 2>/dev/null || true
                    tc qdisc add dev ${ifaces[$i]} root handle 1: pfifo_fast 2>/dev/null || true
                fi"
        done
    fi
) || (echo -e "\nERROR: Failed configuring client nodes\n"; exit 1)}

configure_clients_edge() {(set -e
    local testrate=$1
    local rtt=$2
    local aqm_name=$3
    local aqm_params=$4
    local netem_params=$5  # optional

    configure_clients_edge_aqm_node $testrate $rtt $aqm_name "$aqm_params" "$netem_params"
    configure_clients_node $rtt "$netem_params"
)}

configure_server_edge() {(set -e
    local ip_server_mgmt=$1
    local ip_aqm_s=$2
    local iface_server=$3
    local iface_on_server=$4
    local rtt=$5
    local netem_params=$6  # optional

    local delay=$(echo "scale=2; $rtt / 2" | bc)  # delay is half the rtt

    # put traffic in band 1 by default
    # delay traffic in band 1
    # filter traffic from aqm node itself into band 0 for priority and no delay
    if tc qdisc show dev $iface_server | grep -q 'qdisc netem 12:'; then
        tc qdisc change dev $iface_server handle 12: netem delay ${delay}ms $netem_params
    else
        tc qdisc  del dev $iface_server root 2>/dev/null || true
        tc qdisc  add dev $iface_server root       handle  1: prio bands 2 priomap 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
        tc qdisc  add dev $iface_server parent 1:2 handle 12: netem delay ${delay}ms $netem_params
        tc filter add dev $iface_server parent 1:0 protocol ip prio 1 u32 match ip src $ip_aqm_s flowid 1:1
    fi

    ssh root@$ip_server_mgmt "
        set -e
        if tc qdisc show dev $iface_on_server | grep -q 'qdisc netem 12:'; then
            tc qdisc change dev $iface_on_server handle 12: netem delay ${delay}ms $netem_params
```

```
        else
            tc qdisc  del dev $iface_on_server root 2>/dev/null || true
            tc qdisc  add dev $iface_on_server root       handle 1: prio bands 2 priomap 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
            tc qdisc  add dev $iface_on_server parent 1:2 handle 12: netem delay ${delay}ms $netem_params
            tc filter add dev $iface_on_server parent 1:0 protocol ip prio 1 u32 match ip dst $ip_aqm_s flowid 1:1
        fi"
) || (echo -e "\nERROR: Failed configuring server edge for server $1\n"; exit 1)}

reset_aqm_client_edge() {(set -e
    # reset qdisc at client side
    tc qdisc del dev $IFACE_CLIENTS root 2>/dev/null || true
    tc qdisc add dev $IFACE_CLIENTS root handle 1: pfifo_fast 2>/dev/null || true
)}

reset_aqm_server_edge() {(set -e
    # reset qdisc at server side
    for iface in $IFACE_SERVERA $IFACE_SERVERB; do
        tc qdisc del dev $iface root 2>/dev/null || true
        tc qdisc add dev $iface root handle 1: pfifo_fast 2>/dev/null || true
    done
)}

reset_host() {(set -e
    local host=$1
    local iface=$2 # the iface is the one that test traffic to aqm is going on
                   # e.g. $IFACE_ON_CLIENTA
    ssh root@$host "
        set -e
        tc qdisc del dev $iface root 2>/dev/null || true
        tc qdisc add dev $iface root handle 1: pfifo_fast 2>/dev/null || true"
)}

reset_all_hosts_edge() {(set -e
    hosts=($IP_CLIENTA_MGMT $IP_CLIENTB_MGMT $IP_SERVERA_MGMT $IP_SERVERB_MGMT)
    ifaces=($IFACE_ON_CLIENTA $IFACE_ON_CLIENTB $IFACE_ON_SERVERA $IFACE_ON_SERVERB)

    for i in ${!hosts[@]}; do
        reset_host ${hosts[$i]} ${ifaces[$i]}
    done
)}

reset_all_hosts_cc() {(set -e
    for host in CLIENTA CLIENTB SERVERA SERVERB; do
        name="IP_${host}_MGMT"
        configure_host_cc ${!name} cubic 2
    done
)}

```

```
set_offloading() {(set -e
    onoff=$1

    hosts=($IP_CLIENTA_MGMT $IP_CLIENTB_MGMT $IP_SERVERA_MGMT $IP_SERVERB_MGMT)
    ifaces=($IFACE_ON_CLIENTA $IFACE_ON_CLIENTB $IFACE_ON_SERVERA $IFACE_ON_SERVERB)

    for i in ${!hosts[@]}; do
        ssh root@${hosts[$i]} "
            set -e
            ethtool -K ${ifaces[$i]} gro $onoff
            ethtool -K ${ifaces[$i]} gso $onoff
            ethtool -K ${ifaces[$i]} tso $onoff"
    done

    for iface in $IFACE_CLIENTS $IFACE_SERVERA $IFACE_SERVERB; do
        sudo ethtool -K $iface gro $onoff
        sudo ethtool -K $iface gso $onoff
        sudo ethtool -K $iface tso $onoff
    done
)}

kill_all_traffic() {(set -e
    hosts=($IP_CLIENTA_MGMT $IP_CLIENTB_MGMT $IP_SERVERA_MGMT $IP_SERVERB_MGMT)

    for host in ${hosts[@]}; do
        ssh root@$host '
            set -e
            killall -9 iperf 2>/dev/null || :
            killall -9 greedy 2>/dev/null || :'
    done
)}

get_host_cc() {(set -e
    local host=$1

    # see configure_host_cc for more details on setup

    ssh root@$host '
        set -e
        if [ -f /proc/sys/net/ipv4/tcp_congestion_control ]; then
            sysctl -n net.ipv4.tcp_congestion_control
            sysctl -n net.ipv4.tcp_ecn
        else
            # we are on docker
            . /aqmt-vars-local.sh
            if ip a show $IFACE_AQM | grep -q 10.25.1.; then
                # on client
                route=10.25.2.0/24
            else
                route=10.25.1.0/24
            fi

            ip route show $route | awk -F"congctl " "{print \$2}" | cut -d" " -f1
            ip route show $route | grep -q "ecn" && echo "1" || echo "2"
```

```
        fi'
)}

check_port_in_use() {(set -e
    # output to stdout: 0 if free, or else the number of open sockets
    local host=$1
    local port=$2

    ssh root@$host "
        set -e
        ss -an src :$port | tail -n +2 | wc -l
        "
)}
```

Listing 9: Shell script written to provide functions to configure the testbed.

```
#!/bin/bash
set -e

# we mount ssh setup in a specific template directory
# now we copy this so it is effective
mkdir -p /root/.ssh/
cp /ssh-template/* /root/.ssh/
chown -R root:root /root/.ssh/
chmod 600 /root/.ssh/*

# arp config is done to avoid arp lookups that causes loss

disable_so() {
    iface=$1
    # disable segmentation offload
    # see http://rtodto.net/generic_segmentation_offload_and_wireshark/
    (set -x && ethtool -K $iface gro off)
    (set -x && ethtool -K $iface gso off)
    (set -x && ethtool -K $iface tso off)
}

setup_client() {
    local iface=$(ip route show to 10.25.1.0/24 | awk '{print $3}')
    echo "Adding route to servers through aqm-machine"
    (set -x && ip route add 10.25.2.0/24 via 10.25.1.2 dev $iface)
    (set -x && ip route add 10.25.3.0/24 via 10.25.1.2 dev $iface)
    (set -x && tc qdisc add dev $iface root handle 1: pfifo_fast)
    (set -x && ip link set $iface txqueuelen 1000)
    (set -x && arp -i $iface -s 10.25.1.2 02:42:0a:19:01:02)

    disable_so $iface

    echo "export IFACE_AQM=$iface" >/aqmt-vars-local.sh
}

setup_server() {
```

```
    local iface=$(ip route show to ${1}.0/24 | awk '{print $3}')

    echo "Adding route to clients through aqm-machine"
    (set -x && ip route add 10.25.1.0/24 via ${1}.2 dev $iface)
    (set -x && tc qdisc add dev $iface root handle 1: pfifo_fast)
    (set -x && ip link set $iface txqueuelen 1000)

    disable_so $iface

    (set -x && arp -i $iface -s ${1}.2 02:42:0a:19:0${1/*.}:02)

    echo "export IFACE_AQM=$iface" >/aqmt-vars-local.sh

    #echo "Adding route to other servers through aqm-machine"
    #if [ "$(ip route show to 10.25.2.0/24)" == "" ]; then
    #    (set -x && ip route add 10.25.2.0/24 via ${1}.2 dev $iface)
    #else
    #    (set -x && ip route add 10.25.3.0/24 via ${1}.2 dev $iface)
    #fi
}

setup_aqm() {
    echo "Setting up AQM-variables"

    local iface=$(ip route show to 10.25.0.0/24 | awk '{print $3}')
    echo "export IFACE_MGMT=$iface" >/aqmt-vars-local.sh

    local iface=$(ip route show to 10.25.1.0/24 | awk '{print $3}')
    echo "export IFACE_CLIENTS=$iface" >>/aqmt-vars-local.sh
    (set -x && tc qdisc add dev $iface root handle 1: pfifo_fast)
    (set -x && ip link set $iface txqueuelen 1000)
    (set -x && arp -i $iface -s 10.25.1.11 02:42:0a:19:01:0b)
    (set -x && arp -i $iface -s 10.25.1.12 02:42:0a:19:01:0c)

    disable_so $iface

    local iface=$(ip route show to 10.25.2.0/24 | awk '{print $3}')
    echo "export IFACE_SERVERA=$iface" >>/aqmt-vars-local.sh
    (set -x && tc qdisc add dev $iface root handle 1: pfifo_fast)
    (set -x && ip link set $iface txqueuelen 1000)
    (set -x && arp -i $iface -s 10.25.2.21 02:42:0a:19:02:15)

    disable_so $iface

    local iface=$(ip route show to 10.25.3.0/24 | awk '{print $3}')
    echo "export IFACE_SERVERB=$iface" >>/aqmt-vars-local.sh
    (set -x && tc qdisc add dev $iface root handle 1: pfifo_fast)
    (set -x && ip link set $iface txqueuelen 1000)
    (set -x && arp -i $iface -s 10.25.2.31 02:42:0a:19:03:1f)

    disable_so $iface

```

```
    # wait a bit for other nodes to come up before we try to connect
    sleep 2

    names=(CLIENTA CLIENTB SERVERA SERVERB)
    nets=(10.25.1.0/24 10.25.1.0/24 10.25.2.0/24 10.25.3.0/24)
    for i in ${!names[@]}; do
        (
            . /aqmt-vars.sh
            local ip_name="IP_${names[$i]}"
            local iface
            iface=$(ssh ${!ip_name} "ip route show to ${nets[$i]} | awk '{print \$3}'")
            echo "export IFACE_ON_${names[$i]}=$iface" >>/aqmt-vars-local.sh
        )
    done
}

# add routes through aqm-machine
if [ "$(ip addr show to 10.25.0.2)" == "" ]; then
    if ip a | grep -q "inet 10.25.1."; then
        setup_client
    elif ip a | grep -q "inet 10.25.2."; then
        setup_server 10.25.2
    elif ip a | grep -q "inet 10.25.3."; then
        setup_server 10.25.3
    fi
else
    setup_aqm
fi

echo "Initialization finished"

exec "$@"
```

Listing 10: entrypoint.sh: Initialization script for the Docker containers to configure routing and proper network setup.

A.3 Docker setup

```
version: '2'

services:

  aqm:
    build: .
    image: testbed
    cap_add:
      - NET_ADMIN
    #privileged: true
    hostname: aqm
    networks:
      management:
        ipv4_address: 10.25.0.2
```

```
      clients:
        ipv4_address: 10.25.1.2
      servera:
        ipv4_address: 10.25.2.2
      serverb:
        ipv4_address: 10.25.3.2
    volumes:
      - /etc/hostname:/.dockerhost-hostname # to get real hostname
        ↪ inside docker
      - ../:/opt/aqmt/
      - ./.vars.sh:/aqmt-vars.sh
      - ./container/id_rsa:/ssh-template/id_rsa
      - ./container/id_rsa.pub:/ssh-template/id_rsa.pub
      - ./container/id_rsa.pub:/ssh-template/authorized_keys
      - $TEST_PATH:/opt/testbed

  clienta:
    build: .
    image: testbed
    cap_add:
      - NET_ADMIN
    privileged: true
    hostname: clienta
    networks:
      management:
        ipv4_address: 10.25.0.11
      clients:
        ipv4_address: 10.25.1.11
    volumes:
      - ../:/opt/aqmt/
      - ./.vars.sh:/aqmt-vars.sh
      - ./container/id_rsa:/ssh-template/id_rsa
      - ./container/id_rsa.pub:/ssh-template/id_rsa.pub
      - ./container/id_rsa.pub:/ssh-template/authorized_keys
      - $TEST_PATH:/opt/testbed

  clientb:
    build: .
    image: testbed
    cap_add:
      - NET_ADMIN
    privileged: true
    hostname: clientb
    networks:
      management:
        ipv4_address: 10.25.0.12
      clients:
        ipv4_address: 10.25.1.12
    volumes:
      - ../:/opt/aqmt/
      - ./.vars.sh:/aqmt-vars.sh
      - ./container/id_rsa:/ssh-template/id_rsa
      - ./container/id_rsa.pub:/ssh-template/id_rsa.pub
      - ./container/id_rsa.pub:/ssh-template/authorized_keys
      - $TEST_PATH:/opt/testbed

  servera:
    build: .
    image: testbed
```

```
    cap_add:
      - NET_ADMIN
    privileged: true
    hostname: servera
    networks:
      management:
        ipv4_address: 10.25.0.21
      servera:
        ipv4_address: 10.25.2.21
    volumes:
      - ../:/opt/aqmt/
      - ./.vars.sh:/aqmt-vars.sh
      - ./container/id_rsa:/ssh-template/id_rsa
      - ./container/id_rsa.pub:/ssh-template/id_rsa.pub
      - ./container/id_rsa.pub:/ssh-template/authorized_keys
      - $TEST_PATH:/opt/testbed

  serverb:
    build: .
    image: testbed
    cap_add:
      - NET_ADMIN
    privileged: true
    hostname: serverb
    networks:
      management:
        ipv4_address: 10.25.0.31
      serverb:
        ipv4_address: 10.25.3.31
    volumes:
      - ../:/opt/aqmt/
      - ./.vars.sh:/aqmt-vars.sh
      - ./container/id_rsa:/ssh-template/id_rsa
      - ./container/id_rsa.pub:/ssh-template/id_rsa.pub
      - ./container/id_rsa.pub:/ssh-template/authorized_keys
      - $TEST_PATH:/opt/testbed

  fix_permissions:
    build: .
    image: testbed
    volumes:
      - ../:/opt/aqmt/
      - $TEST_PATH:/opt/testbed
    network_mode: none
    entrypoint: /opt/aqmt/docker/fix-permissions.sh

networks:
  management:
    driver: bridge
    ipam:
      config:
        - subnet: 10.25.0.0/24
  clients:
    driver: bridge
    ipam:
      config:
        - subnet: 10.25.1.0/24
  servera:
    driver: bridge
```

```
      ipam:
        config:
          - subnet: 10.25.2.0/24
    serverb:
      driver: bridge
      ipam:
        config:
          - subnet: 10.25.3.0/24
```

Listing 11: docker-compose.yml: Definition of Docker containers.

```
FROM ubuntu:xenial
MAINTAINER Henrik Steen <henrist@henrist.net>

# set up ssh and custom packages
ADD container/speedometer.patch /opt/
RUN apt-get update \
    && apt-get install -y --no-install-recommends \
         bc \
         ca-certificates \
         dstat \
         ethtool \
         git \
         gnuplot \
         inotify-tools \
         iputils-ping \
         iperf \
         iperf3 \
         iptraf \
         ipython3 \
         less \
         net-tools \
         netcat-openbsd \
         nmap \
         openssh-server \
         patch \
         psmisc \
         python \
         python-urwid \
         python3-numpy \
         python3-plumbum \
         sudo \
         tcpdump \
         tmux \
         vim \
         wget \
    && rm -rf /var/lib/apt/lists/* \
    && mkdir /var/run/sshd \
    \
    && wget -O /usr/bin/speedometer
     ↪   https://raw.githubusercontent.com/wardi/speedometer/9211116e8df11fc6458489b209de2900a
     ↪   \
    && (cd /usr/bin; patch </opt/speedometer.patch) \
    && chmod +x /usr/bin/speedometer \
    \
    # dont check host keys when connecting
```

```
    && sed -i 's/#   StrictHostKeyChecking .*/ StrictHostKeyChecking no/' /etc/ssh/ssh_config \
    \
    # SSH login fix. Otherwise user is kicked off after login
    && sed 's@session\s*required\s*pam_loginuid.so@session optional pam_loginuid.so@g' -i /etc/pam.d/sshd \
    \
    # optimize ssh connection by persisting connection
    && echo "Host 10.25.0.*" >>/etc/ssh/ssh_config \
    && echo "    ControlMaster auto" >>/etc/ssh/ssh_config \
    && echo "    ControlPersist yes" >>/etc/ssh/ssh_config \
    && echo "    ControlPath ~/.ssh/socket-%r@%h:%p" >>/etc/ssh/ssh_config \
    && echo "    AddressFamily inet" >>/etc/ssh/ssh_config \
    \
    && echo ". /aqmt-vars.sh" >>/etc/bash.bashrc \
    && echo "cd /opt/testbed" >>/etc/bash.bashrc \
    && echo ". /aqmt-vars.sh" >>/etc/profile.d/aqmt.sh \
    && echo 'PATH="/opt/aqmt/bin:$PATH"' >>/etc/bash.bashrc \
    && echo 'export PYTHONPATH="/opt/aqmt:$PYTHONPATH"' >>/etc/bash.bashrc

COPY container/iproute2-patches /opt/iproute2-patches
RUN apt-get update \
    && apt-get install -y --no-install-recommends \
         bison \
         build-essential \
         flex \
         git \
         iptables-dev \
         libdb5.3-dev \
         patch \
         pkg-config \
    && rm -rf /var/lib/apt/lists/* \
    \
    # set up custom iproute2 (we need at least v4.6.0 for 'ip route congctl' support
    && cd /opt \
    && git clone --depth=1 --branch=v4.10.0 git://git.kernel.org/pub/scm/linux/kernel/git/shemminger/iproute2.git iproute2 \
    && cd iproute2 \
    && find /opt/iproute2-patches -name "*.patch" -print0 | sort -z | \
         xargs --no-run-if-empty -0 -l patch -p1 -f --fuzz=3 -i \
    && make \
    && make install \
    && cd /opt \
    && rm -rf /opt/iproute2 \
    \
    # set up greedy
    && wget -O /usr/bin/greedy https://github.com/henrist/greedy/releases/download/v0.1/greedy \
    && chmod +x /usr/bin/greedy \
    \
    && apt-get remove -y bison build-essential flex git iptables-dev libdb5.3-dev patch pkg-config \
    && apt-get autoremove -y
```

```

# create a file that can be used to identify we are in Docker
RUN touch /.dockerenv

ADD container/entrypoint.sh /entrypoint.sh

EXPOSE 22
ENTRYPOINT ["/entrypoint.sh"]
CMD ["/usr/sbin/sshd", "-D"]
```

Listing 12: Dockerfile: Definition of Docker image used to run tests.

A.4 Python framework for testing AQMs

This is part of the relevant code developed by me during the thesis. The code for plotting and analyzing is not included. The complete source code of the framework can be found at:

- https://github.com/henrist/aqmt
- https://github.com/henrist/aqmt-example
- https://github.com/henrist/aqmt-fq-codel-scheduler
- https://github.com/henrist/aqmt-pfifo-scheduler
- https://github.com/henrist/aqmt-pie-scheduler

```python
class Testdef:
    def __init__(self, testenv):
        self.collection = None  # set by run_test
        self.dry_run = False  # if dry run no side effects should be caused
        self.post_hook = None
        self.pre_hook = None
        self.testbed = testenv.testbed  # shortcut to above
        self.testenv = testenv
        self.level = 0
        self.test_plots = {
            'analysis': {}, # the value represents **plot_args
        }

    def testcase_analyze(self, testcase, samples_to_skip):
        analyze_test(testcase.test_folder, samples_to_skip)

    def testcase_plot(self, testcase):
        for name, plot_args in self.test_plots.items():
            plot_test(testcase.test_folder, name=name, **plot_args)


def run_test(folder=None, testenv=None, title=None, subtitle=None, steps=None,
        ask_confirmation=None):
```

```python
    """
    Run a complete test using list of steps.

    See steps.py for example steps.
    """
    require_on_aqm_node()
    testdef = Testdef(testenv)

    # Save testdef to testenv so we can pull it from the test case we are running.
    # We use this to hold internal parameters.
    testenv.testdef = testdef

    num_tests = 0
    estimated_time = 0
    num_tests_total = 0

    def get_metadata(testcollection, testenv):
        nonlocal estimated_time, num_tests, num_tests_total
        meta = testcollection.get_metadata(testenv)
        estimated_time += meta['estimated_time'] if meta['will_test'] else 0
        num_tests += 1 if meta['will_test'] else 0
        num_tests_total += 1

    def walk(parent, steps, level=0):
        testdef.collection = parent

        # The last step should be the actual traffic generator
        if len(steps) == 1:
            if testdef.dry_run:
                get_metadata(parent, testenv)
            else:
                parent.run_test(
                    test_fn=steps[0],
                    testenv=testenv,
                    analyze_fn=testdef.testcase_analyze,
                    plot_fn=testdef.testcase_plot,
                    pre_hook=testdef.pre_hook,
                    post_hook=testdef.post_hook,
                )

        else:
            # Each step should be a generator, yielding metadata for new branches.
            # If the generator yields nothing, we jump to next level.
            testdef.level = level
            for step in steps[0](testdef):
                if not step:
                    walk(parent, steps[1:], level)
                    continue

                child = parent
                if len(steps) > 1:
                    child = TestCollection(
                        title=step['title'],
                        titlelabel=step['titlelabel'],
                        folder=step['tag'],
```

```python
                        parent=parent
                    )
                walk(child, steps[1:], level + 1)

                # the walk function have replaced our collection, so put it back
                testdef.collection = parent

    def get_root():
        return TestCollection(
            folder=folder,
            title=title,
            subtitle=subtitle,
        )

    testdef.dry_run = True
    walk(get_root(), steps)
    print('Estimated time: %d seconds for running %d (of %d) tests (average %g sec/test)\n' % (
        estimated_time, num_tests, num_tests_total, estimated_time / num_tests if num_tests > 0 else 0))

    if ask_confirmation is None:
        ask_confirmation = True
        if 'TEST_NO_ASK' in os.environ and os.environ['TEST_NO_ASK'] != '':
            ask_confirmation = False

    should_run_test = not ask_confirmation
    if ask_confirmation:
        sys.stdout.write('Start test? [y/n] ')
        should_run_test = input().lower() == 'y'

    if should_run_test:
        testdef.dry_run = False
        walk(get_root(), steps)
```

Listing 13: aqmt/__init__.py: Python module for running a test definition.

```python
"""
This module contains predefined steps that can be applied
when composing a test hiearchy. Feel free to write your own
instead of using these.

A step is required to yield (minimum one time) in two different ways:

- Yield nothing: This does not cause a branch in the test hierarchy.

- Yield object: Should be an object with the following properties, and
  will cause a new branch with these properties:
  - tag
  - title
```

```python
  - titlelabel
"""

import os.path

from .plot import generate_hierarchy_data_from_folder, \
                  plot_folder_flows, plot_folder_compare, \
                  reorder_levels
from .testcollection import build_html_index

MBIT = 1000*1000


def branch_sched(sched_list, titlelabel='Scheduler'):
    def step(testdef):
        for tag, title, sched_name, sched_params in sched_list:
            testdef.testenv.testbed.aqm(sched_name, sched_params)
            testdef.sched_tag = tag  # to allow substeps to filter it

            yield {
                'tag': 'sched-%s' % tag,
                'title': title,
                'titlelabel': titlelabel,
            }
    return step


def branch_custom(list, fn_testdef, fn_tag, fn_title, titlelabel=''):
    def step(testdef):
        for item in list:
            fn_testdef(testdef, item)
            yield {
                'tag': 'custom-%s' % fn_tag(item),
                'title': fn_title(item),
                'titlelabel': titlelabel,
            }
    return step


def branch_define_udp_rate(rate_list, title='%g', titlelabel='UDP Rate [Mb/s]'):
    """
    This method don't actually change the setup, it only sets a variable
    that can be used when running the actual test.
    """
    def branch(testdef):
        for rate in rate_list:
            testdef.udp_rate = rate
            yield {
                'tag': 'udp-rate-%s' % rate,
                'title': title % rate,
                'titlelabel': titlelabel,
            }
    return branch


```

```python
def branch_repeat(num, title='%d', titlelabel='Test #'):
    def step(testdef):
        for i in range(num):
            yield {
                'tag': 'repeat-%d' % i,
                'title': title % (i + 1),
                'titlelabel': titlelabel,
            }
    return step


def branch_rtt(rtt_list, title='%d', titlelabel='RTT'):
    def step(testdef):
        for rtt in rtt_list:
            testdef.testenv.testbed.rtt_servera = rtt
            testdef.testenv.testbed.rtt_serverb = rtt
            yield {
                'tag': 'rtt-%d' % rtt,
                'title': title % rtt,
                'titlelabel': titlelabel,
            }
    return step


def branch_bitrate(bitrate_list, title='%d', titlelabel='Linkrate [Mb/s]'):
    def step(testdef):
        for bitrate in bitrate_list:
            testdef.testenv.testbed.bitrate = bitrate * MBIT
            yield {
                'tag': 'linkrate-%d' % bitrate,
                'title': title % bitrate,
                'titlelabel': titlelabel,
            }
    return step


def branch_udp_ect(ect_set):
    def branch(testdef):
        for node, ect, title, traffic_tag in ect_set:
            testdef.udp_node = node
            testdef.udp_ect = ect
            testdef.udp_tag = traffic_tag

            yield {
                'tag': 'udp-%s' % ect,
                'title': title,
                'titlelabel': 'UDP ECN',
            }
    return branch


def branch_runif(checks, titlelabel='Run if'):
    def step(testdef):
        for tag, fn, title in checks:
            prev = testdef.testenv.skip_test
            testdef.testenv.skip_test = not fn(testdef.testenv)

            yield {
```

```
                'tag': 'runif-%s' % tag,
                'title': title,
                'titlelabel': titlelabel,
            }

            testdef.testenv.skip_test = prev
    return step


def skipif(fn):
    def step(testdef):
        prev = testdef.testenv.skip_test
        testdef.testenv.skip_test = fn(testdef.testenv)

        yield

        testdef.testenv.skip_test = prev

    return step


def add_pre_hook(fn):
    """
    Add a pre hook to the testcase. Passed to TestCase's run method.
    """
    def step(testdef):
        old_hook = testdef.pre_hook
        def new_hook(*args, **kwargs):
            if callable(old_hook):
                old_hook(*args, **kwargs)
            fn(*args, **kwargs)
        testdef.pre_hook = new_hook
        yield
        testdef.pre_hook = old_hook
    return step


def add_post_hook(fn):
    """
    Add a post hook to the testcase. Passed to TestCase's run method.
    """
    def step(testdef):
        old_hook = testdef.post_hook
        def new_hook(*args, **kwargs):
            if callable(old_hook):
                old_hook(*args, **kwargs)
            fn(*args, **kwargs)
        testdef.post_hook = new_hook
        yield
        testdef.post_hook = old_hook
    return step


def plot_compare(**plot_args):
    def step(testdef):
        yield
```

```
        if not testdef.dry_run and os.path.isdir(testdef.collection.folder):
            plot_folder_compare(testdef.collection.folder, **plot_args)
    return step


def plot_flows(**plot_args):
    def step(testdef):
        yield
        if not testdef.dry_run and os.path.isdir(testdef.collection.folder):
            plot_folder_flows(testdef.collection.folder, **plot_args)
    return step


def plot_test(name='analysis', **plot_args):
    """
    Define a named plot on the test.

    plot_args is sent to plot_test()
    """
    def step(testdef):
        yield
        testdef.test_plots[name] = plot_args
    return step


def html_index(level_order=None):
    def step(testdef):
        yield

        if not testdef.dry_run and os.path.isdir(testdef.collection.folder):
            tree = reorder_levels(
                generate_hierarchy_data_from_folder(testdef.collection.folder),
                level_order=level_order,
            )

            out = build_html_index(tree, testdef.collection.folder)

            with open(testdef.collection.folder + '/analysis.html', 'w') as f:
                f.write(out)

    return step
```

Listing 14: aqmt/steps.py: Python module with components to build the test structure and branching of the test parameters.

```
"""
This module contains the testbed logic
"""
```

```python

import math
import os
from plumbum import local, FG
from plumbum.cmd import bash
from plumbum.commands.processes import ProcessExecutionError

from . import logger
from .terminal import get_log_cmd


def get_testbed_script_path():
    return "aqmt-testbed.sh"


def require_on_aqm_node():
    testbed_script = get_testbed_script_path()
    bash['-c', 'set -e; source %s; require_on_aqm_node' % testbed_script] & FG


class Testbed:
    """
    A object representing the desired testbed configuration and utilities
    to apply the configuration. This object is used throughout tests
    and is mutated and reapplied before tests to change the setup.
    """
    ECN_DISABLED = 0
    ECN_INITIATE = 1
    ECN_ALLOW = 2

    def __init__(self, duration=250*1000, sample_time=1000, idle=None):
        self.bitrate = 1000000

        self.rtt_clients = 0  # in ms
        self.rtt_servera = 0  # in ms
        self.rtt_serverb = 0  # in ms

        self.netem_clients_params = ""
        self.netem_servera_params = ""
        self.netem_serverb_params = ""

        self.aqm_name = 'pfifo_aqmt'  # we need a default aqm to get queue delay
        self.aqm_params = ''

        self.cc_a = 'cubic'
        self.ecn_a = self.ECN_ALLOW
        self.cc_b = 'cubic'
        self.ecn_b = self.ECN_ALLOW

        self.ta_delay = sample_time
        self.ta_samples = math.ceil(duration / sample_time)

        # time to skip in seconds when building aggregated data, default to be RTT-dependent
```

```
        self.ta_idle = idle

        self.traffic_port = 5500

    def aqm(self, name='', params=''):
        if name == 'pfifo':
            name = 'pfifo_aqmt'  # use our custom version with aqmt

        self.aqm_name = name
        self.aqm_params = params

    def cc(self, node, cc, ecn):
        if node != 'a' and node != 'b':
            raise Exception("Invalid node: %s" % node)

        if node == 'a':
            self.cc_a = cc
            self.ecn_a = ecn
        else:
            self.cc_b = cc
            self.ecn_b = ecn

    def rtt(self, rtt_servera, rtt_serverb=None, rtt_clients=0):
        if rtt_serverb is None:
            rtt_serverb = rtt_servera

        self.rtt_clients = rtt_clients  # in ms
        self.rtt_servera = rtt_servera  # in ms
        self.rtt_serverb = rtt_serverb  # in ms

    def get_ta_samples_to_skip(self):
        time = self.ta_idle
        if time is None:
            time = (max(self.rtt_clients, self.rtt_servera, self.rtt_serverb) / 1000) * 40 + 4

        samples = time * 1000 / self.ta_delay
        return math.ceil(samples)

    def setup(self, dry_run=False, log_level=logger.DEBUG):
        cmd = bash['-c', """
            # configuring testbed
            set -e
            source """ + get_testbed_script_path() + """

            set_offloading off

            configure_clients_edge """ + '%s %s %s "%s" "%s"' % (self.bitrate, self.rtt_clients, self.aqm_name, self.aqm_params, self.netem_clients_params) + """
            configure_server_edge $IP_SERVERA_MGMT $IP_AQM_SA $IFACE_SERVERA $IFACE_ON_SERVERA """ + '%s "%s"' % (self.rtt_servera, self.netem_servera_params) + """
            configure_server_edge $IP_SERVERB_MGMT $IP_AQM_SB $IFACE_SERVERB $IFACE_ON_SERVERB """ + '%s "%s"' % (self.rtt_serverb, self.netem_serverb_params) + """

```

```
            configure_host_cc $IP_CLIENTA_MGMT """ + '%s %s' % (self.cc_a, self.ecn_a) + """
            configure_host_cc $IP_SERVERA_MGMT """ + '%s %s' % (self.cc_a, self.ecn_a) + """
            configure_host_cc $IP_CLIENTB_MGMT """ + '%s %s' % (self.cc_b, self.ecn_b) + """
            configure_host_cc $IP_SERVERB_MGMT """ + '%s %s' % (self.cc_b, self.ecn_b) + """
            """]

        logger.log(log_level, get_log_cmd(cmd))
        if not dry_run:
            try:
                cmd & FG
            except ProcessExecutionError:
                return False

        return True

    @staticmethod
    def reset(dry_run=False, log_level=logger.DEBUG):
        cmd = bash['-c', """
            # resetting testbed
            set -e
            source """ + get_testbed_script_path() + """

            kill_all_traffic
            reset_aqm_client_edge
            reset_aqm_server_edge
            reset_all_hosts_edge
            reset_all_hosts_cc
            """]

        logger.log(log_level, get_log_cmd(cmd))
        if not dry_run:
            try:
                cmd & FG
            except ProcessExecutionError:
                return False

        return True

    def get_next_traffic_port(self, node_to_check=None):
        while True:
            tmp = self.traffic_port
            self.traffic_port += 1

            if node_to_check is not None:
                if 'CLIENT' not in node_to_check and 'SERVER' not in node_to_check:
                    raise Exception('Expecting node name like CLIENTA. Got: %s' % node_to_check)
                host = '$IP_%s_MGMT' % node_to_check
                import time
                start = time.time()
                res = bash['-c', """
                    set -e
                    source """ + get_testbed_script_path() + """
```

```
                    check_port_in_use """ + host + """ """ + str(tmp) + """ 2>/dev/null
                    """]()
                if int(res) > 0:
                    # port in use, try next
                    logger.warn('Port %d on node %s was in use - will try next port' % (tmp, node_to_check))
                    continue

            break

        return tmp

    @staticmethod
    def get_aqm_options(name):
        testbed_script = get_testbed_script_path()
        res = bash['-c', 'set -e; source %s; get_aqm_options %s' % (testbed_script, name)]()
        return res.strip()

    def get_setup(self):
        out = ""

        out += "Configured testbed:\n"
        out += "  rate: %s (applied from router to clients)\n" % self.bitrate
        out += "  rtt to router:\n"
        out += "    - clients: %d ms\n" % self.rtt_clients
        out += "    - servera: %d ms\n" % self.rtt_servera
        out += "    - serverb: %d ms\n" % self.rtt_serverb

        if self.aqm_name != '':
            params = ''
            if self.aqm_params != '':
                params = ' (%s)' % self.aqm_params

            out += "  aqm: %s%s\n" % (self.aqm_name, params)
            out += "       (%s)\n" % self.get_aqm_options(self.aqm_name)
        else:
            out += "  no aqm\n"

        for node in ['CLIENTA', 'CLIENTB', 'SERVERA', 'SERVERB']:
            ip = 'IP_%s_MGMT' % node

            out += '  %s: ' % node.lower()
            testbed_script = get_testbed_script_path()
            out += (bash['-c', 'set -e; source %s; get_host_cc "$%s"' % (testbed_script, ip)] | local['tr']['\n', ' '])().strip()
            out += '\n'

        return out.strip()

    def get_hint(self, dry_run=False):
        hint = ''
        hint += "testbed_rtt_clients %d\n" % self.rtt_clients
        hint += "testbed_rtt_servera %d\n" % self.rtt_servera
```

```
        hint += "testbed_rtt_serverb %d\n" % self.rtt_serverb
        hint += "testbed_cc_a %s %d\n" % (self.cc_a, self.ecn_a)
        hint += "testbed_cc_b %s %d\n" % (self.cc_b, self.ecn_b)
        hint += "testbed_aqm %s\n" % self.aqm_name
        hint += "testbed_aqm_params %s\n" % self.aqm_params
        if dry_run:
            hint += "testbed_aqm_params_full UNKNOWN IN DRY RUN\n"
        else:
            hint += "testbed_aqm_params_full %s\n" % self.get_aqm_options(self.aqm_name)
        hint += "testbed_rate %s\n" % self.bitrate
        return hint.strip()
```

Listing 15: aqmt/testbed.py: Python module to define the testbed.

```
"""
Module for utils for plotting collections

See treeutil.py for details of how the tree is structured
"""

from collections import OrderedDict
import math

from .common import PlotAxis
from . import treeutil


def get_tree_details(tree):
    """
    Returns a tuple containing:
    - number of leaf branches
    - number of tests
    - depth of the tree
    - number of x points
    """

    leafs = 0
    tests = 0
    depth = 0
    nodes = 0

    def traverse(branch, depthnow=0):
        nonlocal leafs, tests, depth, nodes

        if len(branch['children']) == 0:
            return

        f = branch['children'][0]

        if depthnow > depth:
            depth = depthnow

        # is this a set of tests?
        if len(f['children']) == 1 and 'testcase' in f['children'][0]:
            tests += len(branch['children'])
```

```
            leafs += 1
            nodes += len(branch['children'])

        # or is it a collection of collections
        else:
            for item in branch['children']:
                nodes += 1
                traverse(item, depthnow + 1)

    traverse(tree)
    return leafs, tests, depth, nodes - depth


def get_gap(tree):
    """
    Calculate the gap that a single test can fill in the graph.
    This tries to make the gap be visually the same for few/many tests.
    """
    _, _, _, n_nodes = get_tree_details(tree)
    return min(0.8, (n_nodes + 2) / 100)


def get_testcases(leaf):
    """
    Get list of testcases of a test collection

    Returns [(title, testcase_folder), ...]
    """
    return [(item['title'], item['children'][0]['testcase']) for item in leaf['children']]


def get_all_testcases_folders(tree):
    """
    Get a list of all testcase folders in a given tree
    """
    folders = []

    def parse_leaf(leaf, first_set, x):
        nonlocal folders
        folders += [item[1] for item in get_testcases(leaf)]  # originally list of (title, folder)

    treeutil.walk_leaf(tree, parse_leaf)
    return folders


def make_xtics(tree, xoffset, x_axis):
    """
    Generate a list of xtics

    This can be passed on to `set xtics add (<here>)` to add xtics
    to the graph.
    """

    arr = []

```

```
    minval, maxval, count = get_testmeta_min_max_count(tree, x_axis)

    numxtics = 10

    def frange(start, stop, step):
        i = start
        while i < stop:
            yield i
            i += step

    #print(minval, maxval)
    #step = ((maxval - minval) / numxtics)
    step = 20 # FIXME: this need to adopt to input
    minval = math.ceil(minval / step) * step
    maxval = math.floor(maxval / step) * step

    #print(minval, maxval)

    for x in frange(minval, maxval + step, step):
        arr.append('"%s" %g' % (
            round(x, 2),
            get_x_coordinate(tree, x, x_axis) + xoffset
        ))

    return ', '.join(arr)


def get_testmeta_min_max_count(leaf, x_axis):
    """
    This function expects all x label titles to be numeric value
    so we can calculate the minimum and maximum of them.
    """
    testcases = get_testcases(leaf)

    # logaritmic, we need to calculate the position
    minval = None
    maxval = None
    for title, testcase_folder in testcases:
        x = float(title)
        if minval is None or x < minval:
            minval = x
        if maxval is None or x > maxval:
            maxval = x

    return minval, maxval, len(testcases)


def get_x_coordinate(leaf, value, x_axis):
    """
    Calculates the linear x position that a value will
    be positioned"""

    minval, maxval, count = get_testmeta_min_max_count(leaf, x_axis)

    pos = float(value)
    if x_axis == PlotAxis.LOGARITHMIC:
        minval = math.log10(minval)
```

```
        maxval = math.log10(maxval)
        pos = math.log10(pos)

    return (pos - minval) / (maxval - minval) * (count - 1) if minval != maxval else 0


def merge_testcase_data_set_x(testcases, x_axis):
    """
    Takes in an array of data points for x axis for a single
    series and appends the x position of the data points.

    Each element in the array is an array itself:
    - xvalue (might be text if linear scale)
    - line (rest of line that is passed on)

    It also concatenates the array and return a final string
    """

    # for category axis we don't calculate anything
    if not PlotAxis.is_logarithmic(x_axis) and not PlotAxis.is_linear(x_axis):
        out = []
        i = 0
        for xval, line in testcases:
            out.append('%d %s' % (i, line))
            i += 1
        return ''.join(out)

    # calculate minimum and maximum value
    minval = None
    maxval = None
    for xval, line in testcases:
        x = float(xval)
        if minval is None or x < minval:
            minval = x
        if maxval is None or x > maxval:
            maxval = x

    if PlotAxis.is_logarithmic(x_axis):
        minval = math.log10(minval)
        maxval = math.log10(maxval)

    out = []
    for xval, line in testcases:
        pos = float(xval)
        if PlotAxis.is_logarithmic(x_axis):
            pos = math.log10(pos)
        x = (pos - minval) / (maxval - minval) * (len(testcases) - 1) if maxval != minval else 0
        out.append('%f %s' % (x, line))
    return ''.join(out)


def get_leaf_tests_stats(leaf, statsname):
    """
    Build data for a specific statistic from all testcases in a leaf

```

```
    The return value will be a list of tupples where first
    element is the title of this test and second element is
    the lines from the statistics with the title appended.
    """
    res = []
    for title, testcase_folder in get_testcases(leaf):
        added = False

        if callable(statsname):
            for line in statsname(testcase_folder).splitlines():
                if line.startswith('#'):
                    continue

                res.append((title, '"%s" %s\n' % (title, line)))
                added = True
                break  # only allow one line from each sample

        else:
            with open(testcase_folder + '/' + statsname, 'r') as f:
                for line in f:
                    if line.startswith('#'):
                        continue

                    res.append((title, '"%s" %s' % (title, line)))
                    added = True
                    break  # only allow one line from each sample

        if not added:
            res.append((title, '"%s"' % title))

    return res


def merge_testcase_data(leaf, statsname, x_axis):
    """
    statsname might be a function. It will be given the folder path
      of the test case and should return one line.
    """
    res = get_leaf_tests_stats(leaf, statsname)
    return merge_testcase_data_set_x(res, x_axis)


def merge_testcase_data_group(leaf, statsname, x_axis):
    """
    Similar to merge_testcase_data except it groups all data by first column

    There should only exist one data point in the files for each group
    """
    out = OrderedDict()

    i_file = 0
    for title, testcase_folder in get_testcases(leaf):
        with open(testcase_folder + '/' + statsname, 'r') as f:
            for line in f:
                if line.startswith('#') or line == '\n':
```

```
                    continue

                if line.startswith('"'):
                    i = line.index('"', 1)
                    group_by = line[1:i]
                else:
                    group_by = line.split()[0]

                if group_by not in out:
                    out[group_by] = [[title, '"%s"\n' % title]] * i_file

                out[group_by].append([title, '"%s" %s' % (title, line)])

        i_file += 1
        for key in out.keys():
            if len(out[key]) != i_file:
                out[key].append([title, '"%s"\n' % title])

    for key in out.keys():
        out[key] = merge_testcase_data_set_x(out[key], x_axis)

    return out


def get_xlabel(tree):
    """
    Get the label that are used for the x axis.

    This is taken from the titlelabel of a test collection.
    """
    xlabel = None

    def fn(leaf, first_set, x):
        nonlocal xlabel
        if xlabel is None and len(leaf['children']) > 0 and leaf['children'][0]['titlelabel'] != '':
            xlabel = leaf['children'][0]['titlelabel']

    treeutil.walk_leaf(tree, fn)
    return xlabel


def pt_generator():
    """
    Building point styles for use in plots.
    """
    pool = [1,2,3,8,10,12,14]
    i = 0
    tags = {}

    def get_val(tag):
        nonlocal i
        if tag not in tags:
            tags[tag] = pool[i % len(pool)]
            i += 1

        return tags[tag]
```

```python

    return get_val
```

Listing 16: aqmt/plot/collectionutil.py: Python module with utilities for plotting a collection/tree.

```python
"""
Module for manipulating the tree structure of collections

Definitions:
- Tree: the root node being handled)
- Branch: a branch inside the tree that contains other collections)
- Leaf branch: the last branch that contains test collections

Example tree:

    Abstract view:

                             root                 ("root", "tree", "branch", "collection")
                         /          \             (root contains plot title)
                         |          |
                    (possible more levels)        ("branch", "collection")
                        /            \
     linkrate:    10 mbit            20 mbit      ("branch", "leaf branch", "collection")
                 /       \          /       \
     rtt:      2 ms     10 ms     2 ms     10 ms  ("collection", "leaf collection")
                |        |         |        |     (only one collection inside leaf branches)
                |        |         |        |
               test     test      test     test   ("test")
                                                 (only one test in leaf collections)

    The reason for having tests as children similar as normal
    branches is to allow easy manipulation of the tree, e.g.
    swapping levels.

    Actual structure:

    {
        'title': 'Plot title',
        'titlelabel': '',
        'subtitle': '',
        'children': [
            {
                'title': '10 Mb/s',
                'titlelabel': 'Linkrate',
                'subtitle': '',
                'children': [
                    {
```

```
                        'title': '2',
                        'titlelabel': 'RTT',
                        'subtitle': '',
                        'children': [
                            {'testcase': 'results/plot-tree/linkrate-10/rtt-2/test'}
                        ],
                    },
                    {
                        'title': '10',
                        'titlelabel': 'RTT',
                        'subtitle': '',
                        'children': [
                            {'testcase': 'results/plot-tree/linkrate-10/rtt-10/test'}
                        ],
                    },
                ],
            },
            {
                'title': '20 Mb/s',
                'titlelabel': 'Linkrate',
                'subtitle': '',
                'children': [
                    {
                        'title': '2',
                        'titlelabel': 'RTT',
                        'subtitle': '',
                        'children': [
                            {'testcase': 'results/plot-tree/linkrate-20/rtt-2/test'}
                        ]
                    },
                    {
                        'title': '10',
                        'titlelabel': 'RTT',
                        'subtitle': '',
                        'children': [
                            {'testcase': 'results/plot-tree/linkrate-20/rtt-10/test'}
                        ]
                    },
                ],
            },
        ],
    }

X offsets:
    X offsets in the tree are increased so that they cause natural
    gaps betweep test branches. So between branches at a deep level
    there is a small gap, while close to the root branch there will
    be more gap.

    In the example above the tests would have the following x offsets
      - test 1: 0
      - test 2: 1
```

```python
      - test 3: 3 (new branch, so x is increased to form a gap)
      - test 4: 4
"""

from collections import OrderedDict


def get_depth_sizes(tree):
    """
    Calculate the number of branches at each tree level
    """
    depths = {}

    def check_node(item, x, depth):
        if depth not in depths:
            depths[depth] = 0
        depths[depth] += 1

    walk_tree(tree, check_node)
    return depths


def walk_leaf(tree, fn):
    """
    Walks the tree and calls fn for every leaf branch

    The arguments to fn:
    - object: the leaf branch
    - bool: true if first leaf branch in tree
    - number: the x offset of this leaf branch
    """

    x = 0
    is_first = True

    def walk(branch):
        nonlocal is_first, x

        if len(branch['children']) == 0:
            return

        first_child = branch['children'][0]

        is_leaf_branch = 'testcase' in branch['children'][0]['children'][0]
        if is_leaf_branch:
            fn(branch, is_first, x)
            is_first = False
            x += len(branch['children'])

        # or is it a collection of collections
        else:
            for item in branch['children']:
                walk(item)

        x += 1

    walk(tree)

```

```python

def walk_tree_reverse(tree, fn):
    """
    Walks the tree and calls fn for every branch in reverse order

    The arguments to fn:
    - object: the branch
    - number: the x offset of this branch
    - number: depth of this branch, 0 being root
    - number: the number of tests inside this branch
    """
    x = 0

    def walk(branch, depth=0):
        nonlocal x

        is_leaf_branch = 'testcase' in branch['children'][0]['children'][0]
        if is_leaf_branch:
            x += len(branch['children'])

        # or else it is a non-leaf branch
        else:
            for item in branch['children']:
                y = x
                walk(item, depth + 1)
                fn(item, y, depth, x - y)

        x += 1

    walk(tree, 0)


def walk_tree(tree, fn, include_leaf_collection=False):
    """
    Walks the tree and calls fn for every branch, and also for every
    leaf collection if include_leaf_collection is True.

    The arguments given to fn:
    - object: the collection
    - number: the x offset related to number of tests/levels
    - number: depth of this collection, 0 being root
    """
    x = 0

    def walk(collection, depth=0):
        nonlocal x

        for subcollection in collection['children']:
            fn(subcollection, x, depth)

            if include_leaf_collection:
                is_leaf_collection = 'testcase' in subcollection['children'][0]
                if is_leaf_collection:
                    x += 1
                    continue

```

```
            # If input to walk_tree was a leaf branch, we can't
            ↪ look
            # if we have  leaf branch inside
            elif 'children' not in subcollection['children'][0]:
                continue

            else:
                is_leaf_branch = 'testcase' in
                ↪ subcollection['children'][0]['children'][0]
                if is_leaf_branch:
                    x += len(subcollection['children']) + 1
                    continue

            walk(subcollection, depth + 1)

        x += 1

    walk(tree)


def swap_levels(tree, level=0):
    """
    Rearrange vertical position of elements in the tree.

    This swaps collections in the tree so their level
    in the tree is changed.

    For the plotting, this will change the way tests
    are grouped and presented.
    """

    if level > 0:
        def walk(branch, depth):
            if len(branch['children']) == 0:
                return

            # is this a set of tests?
            if 'testcase' in branch['children'][0]:
                return

            for index, item in enumerate(branch['children']):
                if depth + 1 == level:
                    branch['children'][index] = swap_levels(item)
                else:
                    walk(item, depth + 1)

        walk(tree, 0)
        return tree

    titles = []

    def check_level(node, x, depth):
        nonlocal titles
        if depth == 1 and node['title'] not in titles:
            titles.append(node['title'])

    walk_tree(tree, check_level, include_leaf_collection=True)

    if len(titles) == 0:
```

```
        return tree

    new_children = OrderedDict()
    parent = None

    def build_swap(node, x, depth):
        nonlocal parent, new_children
        if depth == 0:
            parent = node
        elif depth == 1:
            parentcopy = dict(parent)
            if node['title'] in new_children:
                new_children[node['title']]['children'].append(parentcopy)
            else:
                childcopy = dict(node)
                childcopy['children'] = [parentcopy]
                new_children[node['title']] = childcopy

            parentcopy['children'] = node['children']

    walk_tree(tree, build_swap, include_leaf_collection=True)

    tree['children'] = [val for key, val in new_children.items()]
    return tree


def build_swap_list(level_order):
    """
    Build a list of levels that should be swapped to achieve
    a specific ordering of levels.
    """

    # assert the values
    distinct = []
    for val in level_order:
        if val in distinct:
            raise Exception("Duplicate value: %s" % val)
        if not isinstance(val, int):
            raise Exception("Invalid type: %s" % val)
        if val < 0:
            raise Exception("Value out of bounds: %s" % val)
        distinct.append(val)

    # fill any missing values
    for i in range(max(level_order)):
        if i not in level_order:
            level_order.append(i)

    # work through the list and build a swap list
    swap_list = []
    to_process = list(range(len(level_order)))  # same as an sorted version of the list
    for i in range(len(level_order)):
        # find offset of this target
        to_swap = 0
        while level_order[i] != to_process[to_swap]:
            to_swap += 1

```

```python
        # pull up the target so it become the current level
        for x in range(to_swap):
            swap_list.append(i + (to_swap - x - 1))

        # remove the level we targeted
        to_process.remove(level_order[i])

    return swap_list


def reorder_levels(tree, level_order=None):
    """
    Order the tree based on an ordering of levels
    (number of branches in height in the tree)

    E.g. a tree of 3 levels where we want to reorder the levels
    so that the order is last level, then the first and then the
    second:

      level_order=[2,0,1]

    Example reversing the order of three levels:

      level_order=[2,1,0]
    """

    if level_order is None or len(level_order) == 0:
        return tree

    # get the depth of the tree only counting branches
    levels = len(get_depth_sizes(tree))

    swap_list = build_swap_list(level_order)
    if len(swap_list) > 0 and max(swap_list) >= levels:
        raise Exception("Out of bound level: %d. Only have %d levels" % (max(swap_list), levels))

    # apply the calculated node swapping to the tree
    for level in swap_list:
        tree = swap_levels(tree, level)

    return tree


def skip_levels(tree, number_of_levels):
    """
    Select the left node number_of_levels deep and
    return the new tree
    """

    # allow to select specific branches in a three instead of default first
    if type(number_of_levels) is list:
        for branch in number_of_levels:
            tree = tree['children'][branch]
        return tree

    while number_of_levels > 0:
        tree = tree['children'][0]
```

```
        number_of_levels -= 1

    return tree
```

Listing 17: aqmt/plot/treeutil.py: Python module for manipulating the tree structure of collections from a test.

```
def generate_hierarchy_data_from_folder(folder):
    """
    Generate a dict that can be sent to CollectionPlot by analyzing the directory

    It will look in all the metadata stored while running test
    to generate the final result
    """

    def parse_folder(subfolder):
        if not os.path.isdir(subfolder):
            raise Exception('Non-existing directory: %s' % subfolder)

        metadata_kv, metadata_lines = read_metadata(subfolder + '/details')

        if 'type' not in metadata_kv:
            raise Exception('Missing type in metadata for %s' % subfolder)

        if metadata_kv['type'] in ['collection']:
            node = {
                'title': metadata_kv['title'] if 'title' in metadata_kv else '',
                'subtitle': metadata_kv['subtitle'] if 'subtitle' in metadata_kv else '',
                'titlelabel': metadata_kv['titlelabel'] if 'titlelabel' in metadata_kv else '',
                'children': []
            }

            for metadata in metadata_lines:
                if metadata[0] == 'sub':
                    node['children'].append(parse_folder(subfolder + '/' + metadata[1]))

        elif metadata_kv['type'] == 'test':
            node = {
                'testcase': subfolder
            }

        else:
            raise Exception('Unknown metadata type %s' % metadata_kv['type'])

        return node

    root = parse_folder(folder)
    return root
```

Listing 18: generate_hierarchy_data_from_folder(): Python function for reconstructing the test definition used at test time into a tree representing the test.

A.5 Test code

```python
#!/usr/bin/env python3
#
# Test for simple overloading using only UDP traffic.
#

import math
import sys
import time

from aqmt import MBIT, Testbed, TestEnv, archive_test, run_test, steps
from aqmt.plot import collection_components, flow_components
from aqmt.traffic import udp

import _plugin_tc


def test(result_folder):
    def my_test(testcase):
        testdef = testcase.testenv.testdef

        # split UDP rate in multiple connections
        # to avoid high rates being limited
        udp_rate = testdef.udp_rate
        to_kill = []
        while udp_rate > 0:
            this_rate = min(200, udp_rate)
            udp_rate -= this_rate

            flow = testcase.traffic(
                udp,
                node=testdef.udp_node,
                bitrate=this_rate * MBIT,
                ect=testdef.udp_ect,
                tag=testdef.udp_tag,
            )
            to_kill.append(flow)

        time.sleep(30)
        for flow in to_kill:
            flow()

    testbed = Testbed(40*1000, 250, idle=0)
    testbed.rtt(0)  # doesn't really matter for UDP-only test

    level_order = [
        1,  # bitrate
        0,  # scheduler
```

```
        2,  # udp rate
        3,  # udp queue
    ]

    archive_test(__file__, result_folder)

    run_test(
        folder=result_folder,
        title='Overload testing with only UDP',
        testenv=TestEnv(testbed, replot=True),
        steps=(
            steps.add_pre_hook(_plugin_tc.pre_hook),
            steps.html_index(level_order=level_order),
            steps.plot_compare(level_order=level_order, components=[
                collection_components.utilization_total_only(),
                collection_components.queueing_delay(),
                collection_components.drops_marks(),
                _plugin_tc.plot_comparison_prob(),
                _plugin_tc.plot_comparison_backlog_pkts(),
            ]),
            steps.plot_test(name='thesis', title=None, components=[
                flow_components.queueing_delay(),
                flow_components.queueing_delay(range_to='40'),
                _plugin_tc.plot_flow_prob(),
                                _plugin_tc.plot_flow_backlog_pkts(y_logarithmic=True),
            ], skip_sample_line=True, x_scale=0.6, y_scale=0.6),
            steps.plot_test(components=[
                flow_components.utilization_queues(),
                flow_components.rate_per_flow(),
                flow_components.queueing_delay(),
                flow_components.drops_marks(),
                _plugin_tc.plot_flow_prob(),
                _plugin_tc.plot_flow_backlog_pkts(),
            ]),
            steps.branch_sched([
                # tag, title, name, params
                ('dualpi2-1000',
                    'dualpi2 1000p',
                    'dualpi2', 'dc_dualq dc_ecn target 15ms tupdate 15ms alpha 5 beta 50 k 2 t_shift 30ms l_drop 100 limit 1000'),
                ('pie-1000', 'PIE 1000p', 'pie', 'ecn target 15ms tupdate 15ms alpha 1 beta 10 limit 1000'),
                ('dualpi2-10000',
                    'dualpi2 10000p',
                    'dualpi2', 'dc_dualq dc_ecn target 15ms tupdate 15ms alpha 5 beta 50 k 2 t_shift 30ms l_drop 100 limit 10000'),
                ('pie-10000', 'PIE 10000p', 'pie', 'ecn target 15ms tupdate 15ms alpha 1 beta 10 limit 10000'),
                #('pie-def', 'PIE default', 'pie', 'ecn'),
            ]),
            steps.branch_bitrate([
                100,
                300,
```

```
                500,
            ]),
            steps.branch_define_udp_rate([
                #50,
                100,
                200,
                400,
                800,
            ]),
            steps.branch_udp_ect([
                # node, flag, title, traffic tag
                ['a', 'nonect', 'Non-ECT', 'UDP=Non ECT'],
                ['b', 'ect1', 'ECT(1)', 'UDP=ECT(1)'],
            ]),
            my_test,
        )
    )

if __name__ == '__main__':
    if len(sys.argv) < 2:
        print('Provide an argument for where to store results')
        sys.exit(1)

    test(sys.argv[1])
```

Listing 19: overload-simple.py

```
#!/usr/bin/env python3

import math
import sys
import time

from aqmt import MBIT, Testbed, TestEnv, archive_test, run_test,
 ↪  steps
from aqmt.plot import PlotAxis, collection_components,
 ↪  flow_components
from aqmt.plugins import dstat, ss_rtt
from aqmt.traffic import greedy, udp

import _plugin_tc


def test(result_folder):

    def custom_cc(testdef):
        testdef.testbed.cc('a', 'cubic', testbed.ECN_ALLOW)
        testdef.flows_a_tag = 'CUBIC (no ECN)'
        testdef.flows_a_title = 'C'
        if testdef.testbed.aqm_name in ['pi2', 'dualpi2']:
            testdef.testbed.cc('b', 'dctcp-drop',
             ↪  testbed.ECN_INITIATE)
            testdef.flows_b_tag = 'DCTCP (ECN)'
            testdef.flows_b_title = 'D'
        else:
            testdef.testbed.cc('b', 'cubic', testbed.ECN_INITIATE)
            testdef.flows_b_tag = 'ECN-CUBIC'
```

```
        testdef.flows_b_title = 'EC'

    # no yield value as we don't cause a new branch
    yield

def branch_flow_set(flow_list):
    def branch(testdef):
        for flows_a_num, flows_b_num in flow_list:
            testdef.flows_a_num = flows_a_num
            testdef.flows_b_num = flows_b_num
            yield {
                'tag': 'flow-%d-%d' % (flows_a_num, flows_b_num),
                #'title': '%d x %s vs %d x %s' % (
                #    flows_a_num,
                #    testdef.flows_a_title,
                #    flows_b_num,
                #    testdef.flows_b_title,
                #),
                'title': '%d Non-ECN vs %d ECN' % (
                    flows_a_num,
                    flows_b_num,
                ),
                'titlelabel': 'Flow combination',
            }
    return branch

def my_test(testcase):
    testdef = testcase.testenv.testdef

    for x in range(testdef.flows_a_num):
        testcase.traffic(greedy, node='a', tag=testdef.flows_a_tag)

    for x in range(testdef.flows_b_num):
        testcase.traffic(greedy, node='b', tag=testdef.flows_b_tag)

    if testdef.udp_rate > 0:
        time.sleep(1)
        testcase.traffic(
            udp,
            node=testdef.udp_node,
            bitrate=testdef.udp_rate * MBIT,
            ect=testdef.udp_ect,
            tag=testdef.udp_tag,
        )

testbed = Testbed()

testbed.bitrate = 100 * MBIT

t = 20
#testbed.ta_idle = 0
testbed.ta_delay = 250
testbed.ta_samples = math.ceil(t / (testbed.ta_delay/1000))

archive_test(__file__, result_folder)
```

```python
    level_order = [
        3,  # udp queue
        0,  # rtt
        2,  # flow combination
        1,  # shed
        4,  # udp rate
    ]

    run_test(
        folder=result_folder,
        title='Overload in mixed traffic',
        subtitle='Testrate: 100 Mb/s - D = DCTCP, C = CUBIC, EC = ECN-CUBIC',
        testenv=TestEnv(testbed, retest=False, reanalyze=False),
        steps=(
            steps.add_pre_hook(_plugin_tc.pre_hook),
            steps.add_pre_hook(dstat.pre_hook),
            steps.add_pre_hook(ss_rtt.pre_hook),
            steps.add_pre_hook(lambda testcase: time.sleep(2)),
            steps.html_index(level_order=level_order),
            steps.plot_compare(level_order=level_order, x_axis=PlotAxis.LOGARITHMIC, components=[
                collection_components.utilization_tags(),
                collection_components.queueing_delay(),
                collection_components.drops_marks(),
                dstat.plot_comparison_cpu(),
                dstat.plot_comparison_int_csw(),
                ss_rtt.plot_comparison_rtt(),
                _plugin_tc.plot_comparison_prob(),
                _plugin_tc.plot_comparison_backlog_pkts(),
            ], lines_at_x_offset=[100], x_scale=3), # 3
            steps.plot_compare(
                name='thesis-nonect-a', title=False, subtitle=False,
                level_order=level_order, x_axis=PlotAxis.LOGARITHMIC,
                skip_levels=[0, 0],
                components=[
                    collection_components.utilization_tags(),
                    collection_components.queueing_delay(range_to='40'),
                    collection_components.drops_marks(),
                ],
                lines_at_x_offset=[100],
                x_scale=0.7, y_scale=1,
            ),
            steps.plot_compare(
                name='thesis-nonect-b', title=False, subtitle=False,
                level_order=level_order, x_axis=PlotAxis.LOGARITHMIC,
                skip_levels=[0, 0],
                components=[
                    collection_components.utilization_tags(y_logarithmic=True,
                        range_from_log='0.01', range_to_log='10'),
                    ss_rtt.plot_comparison_rtt(subtract_base_rtt=True, keys=False),
```

```
                    _plugin_tc.plot_comparison_prob(),
                    _plugin_tc.plot_comparison_backlog_pkts(),
                ],
                lines_at_x_offset=[100],
                x_scale=0.7, y_scale=0.7,
            ),
            steps.plot_compare(
                name='thesis-ect1-a', title=False, subtitle=False,
                level_order=level_order, x_axis=PlotAxis.LOGARITHMIC,
                skip_levels=[1, 0],
                components=[
                    collection_components.utilization_tags(),
                    collection_components.queueing_delay(range_to='40'),
                    collection_components.drops_marks(),
                ],
                lines_at_x_offset=[100],
                x_scale=0.7, y_scale=1,
            ),
            steps.plot_compare(
                name='thesis-ect1-b', title=False, subtitle=False,
                level_order=level_order, x_axis=PlotAxis.LOGARITHMIC,
                skip_levels=[1, 0],
                components=[
                    collection_components.utilization_tags(y_logarithmic=True,
                        range_from_log='0.01', range_to_log='10'),
                    ss_rtt.plot_comparison_rtt(subtract_base_rtt=True, keys=False),
                    _plugin_tc.plot_comparison_prob(),
                    _plugin_tc.plot_comparison_backlog_pkts(),
                ],
                lines_at_x_offset=[100],
                x_scale=0.7, y_scale=0.7,
            ),
            steps.plot_test(components=[
                flow_components.utilization_queues(),
                flow_components.rate_per_flow(),
                flow_components.rate_per_flow(y_logarithmic=True),
                flow_components.queueing_delay(),
                flow_components.queueing_delay(y_logarithmic=True),
                flow_components.drops_marks(),
                flow_components.drops_marks(y_logarithmic=True),
                dstat.plot_flow_cpu(),
                dstat.plot_flow_int_csw(),
                ss_rtt.plot_flow_rtt(initial_delay=2),
                _plugin_tc.plot_flow_prob(initial_delay=2),
                _plugin_tc.plot_flow_backlog_pkts(initial_delay=2),
            ]),
            steps.branch_rtt([
                #2,
                10,
                #50,
            ]),
```

```
            steps.branch_sched([
                # tag, title, name, params
                ('dualpi2',
                    'DualPI2',
                    'dualpi2', 'dc_dualq dc_ecn target 15ms tupdate 15ms alpha 5 beta 50 k 2 t_shift 30ms l_drop 100'),
                ('pie', 'PIE', 'pie', 'ecn target 15ms tupdate 15ms alpha 1 beta 10'),
                #('fq_codel', 'FQ-CoDel', 'fq_codel', 'ecn'),
                #('pfifo', 'pfifo', 'pfifo', ''),
            ]),
            custom_cc,
            branch_flow_set([
                # num normal in a, num normal in b
                [0, 1],
                [1, 0],
                #[1, 1],
                #[1, 2],
                #[2, 1],
                [5, 5],
                #[10, 10],
            ]),
            steps.branch_udp_ect([
                # node, flag, title, traffic tag
                ['a', 'nonect', 'Non-ECT', 'UDP=Non ECT'],
                ['b', 'ect1', 'ECT(1)', 'UDP=ECT(1)'],
            ]),
            #steps.plot_flows(),
            steps.branch_define_udp_rate([x + 0 for x in [
                70,
                80,
                90,
                93,
                95,
                96,
                97,
                97.5,
                98,
                98.5,
                99,
                99.5,
                100,
                100.5,
                101,
                102,
                103,
                104,
                105,
                106,
                107,
                108,
                109,
                110,
                111,
                112,
                113,
                114,
                115,
```

```
                116,
                117,
                118,
                119,
                120,
                121,
                122,
                123,
                124,
                125,
                128, ##
                130, ##
                135, ##
                140, ##
                150,
                160, ##
                180, ##
                200,
            ]], title='%d'),
            my_test,
        ),
    )

if __name__ == '__main__':
    if len(sys.argv) < 2:
        print('Provide an argument for where to store results')
        sys.exit(1)

    test(sys.argv[1])
```

Listing 20: overload-mixed.py

```
#!/usr/bin/env python3
#
# Test for testing the rate we can achieve using DCTCP
# on DualPI2.
#

import math
import sys
import time

from aqmt import MBIT, Testbed, TestEnv, archive_test, run_test, steps
from aqmt.plot import PlotAxis, collection_components, flow_components
from aqmt.plugins import ss_rtt
from aqmt.traffic import greedy, udp

def test(result_folder):

    def branch_num_flows(flow_set):
        def branch(testdef):
            for num in flow_set:
                testdef.num_flows = num

                yield {
```

```
                        'tag': 'num-flows-%s' % num,
                        'title': num,
                        'titlelabel': '# flows',
                    }
        return branch

    def set_idle(testdef):
        testbed = testdef.testenv.testbed
        est_window = (testbed.rtt_servera / 1000) *
          ↪ testbed.bitrate / 8 / 1448
        inc_per_sec = 1000 / (testbed.rtt_servera + 2)

        testbed.ta_idle = 1 + est_window / inc_per_sec * 1.5
        yield

    def skip_large_window(testenv):
        limit = 900 * 1448 * 8
        rtt = testenv.testbed.rtt_servera / 1000
        if rtt * testenv.testbed.bitrate > limit:
            return True

        #est_window = rtt * testbed.bitrate / 8 / 1448
        #inc_per_sec = 1000 / (testbed.rtt_servera + 2)
        #if est_window / inc_per_sec > 15:
        #     print(est_window / inc_per_sec)
        #     return True

        return False


    def my_test(testcase):
        for x in range(testcase.testenv.testdef.num_flows):
            testcase.traffic(greedy, node='a', tag='DCTCP')

    testbed = Testbed(10*1000, 250)
    testbed.cc('a', 'dctcp-drop', testbed.ECN_INITIATE)

    level_order = [
        3,  # num flows
        0,  # threshold
        1,  # bitrate
        2,  # rtt
    ]

    archive_test(__file__, result_folder)

    run_test(
        folder=result_folder,
        title='Testing DCTCP rate on DualPI2',
        testenv=TestEnv(testbed),
        steps=(
            steps.add_pre_hook(ss_rtt.pre_hook),
            steps.html_index(level_order=level_order),
            steps.plot_compare(level_order=level_order,
              ↪ components=[
                collection_components.utilization_queues(),
                collection_components.utilization_tags(),

                  ↪ collection_components.window_rate_ratio(y_logarithmic=True),
```

```
            collection_components.window_rate_ratio(),
            collection_components.queueing_delay(keys=False),
            collection_components.drops_marks(),
            ss_rtt.plot_comparison_rtt(subtract_base_rtt=True, keys=False),
        ]),
        steps.plot_test(name='thesis', title=None, components=[
            flow_components.utilization_queues(ecn=False, flows=False),
            flow_components.window(),
            flow_components.queueing_delay(),
            flow_components.drops_marks(show_total=False),
            ss_rtt.plot_flow_rtt(subtract_base_rtt=True),
        ], x_scale=0.5, y_scale=0.5),
        steps.plot_test(components=[
            flow_components.utilization_queues(),
            flow_components.rate_per_flow(),
            flow_components.rate_per_flow(y_logarithmic=True),
            flow_components.window(),
            flow_components.window(y_logarithmic=True),
            flow_components.queueing_delay(),
            flow_components.queueing_delay(y_logarithmic=True),
            flow_components.drops_marks(),
            flow_components.drops_marks(y_logarithmic=True),
            ss_rtt.plot_flow_rtt(initial_delay=2, subtract_base_rtt=True),
        ]),
        steps.branch_sched([
            # tag, title, name, params
            #('dualpi2-500',
            #    '0.5',
            #    'dualpi2', 'dc_dualq dc_ecn target 15ms tupdate 15ms alpha 5 beta 50 k 2 t_shift 30ms l_drop 100 l_thresh 500'),
            ('dualpi2-1000',
                '1',
                'dualpi2', 'dc_dualq dc_ecn target 15ms tupdate 15ms alpha 5 beta 50 k 2 t_shift 30ms l_drop 100 l_thresh 1000'),
            #('dualpi2-2000',
            #    '2',
            #    'dualpi2', 'dc_dualq dc_ecn target 15ms tupdate 15ms alpha 5 beta 50 k 2 t_shift 30ms l_drop 100 l_thresh 2000'),
            #('dualpi2-3000',
            #    '3',
            #    'dualpi2', 'dc_dualq dc_ecn target 15ms tupdate 15ms alpha 5 beta 50 k 2 t_shift 30ms l_drop 100 l_thresh 3000'),
            ('dualpi2-5000',
                '5',
                'dualpi2', 'dc_dualq dc_ecn target 15ms tupdate 15ms alpha 5 beta 50 k 2 t_shift 30ms l_drop 100 l_thresh 5000'),
            #('dualpi2-10000',
            #    '10',
```

```
                #    'dualpi2', 'dc_dualq dc_ecn target 15ms tupdate 15ms alpha 5 beta 50 k 2 t_shift 30ms l_drop 100 l_thresh 10000'),
            ], titlelabel='Threshold [ms]'),
            steps.plot_compare(
                name='thesis', x_axis=PlotAxis.LINEAR_XTICS,
                title=False, level_order=[2,0,1], skip_levels=1, components=[
                    collection_components.utilization_total_only(),
                    collection_components.queueing_delay(keys=False),
                    collection_components.window(keys=False),
                    collection_components.drops_marks(),
                    ss_rtt.plot_comparison_rtt(subtract_base_rtt=True, keys=False),
                ], x_scale=0.7, y_scale=0.6
            ),
            steps.branch_bitrate([
                10,
                50,
                100,
                200,
                400,
                #800,
            ]),
            steps.branch_rtt([
                1,
                2,
                5,
                8,
                10,
                12,
                14,
                15,
                17,
                20,
                25,
                30,
            ]),
            branch_num_flows([
                1,
                #2,
                #3,
                #15,
                #30,
            ]),
            steps.skipif(skip_large_window),
            set_idle,
            my_test,
        )
    )

if __name__ == '__main__':
    if len(sys.argv) < 2:
        print('Provide an argument for where to store results')
        sys.exit(1)

```

```
    test(sys.argv[1])
```

Listing 21: Test for threshold of ultra-low delay. See chapter 13.

Appendix B

‘Data Centre to the Home’: Deployable Ultra-Low Queuing Delay for All

The paper is included as an appendix as it is still under submission and not yet published.

‘Data Centre to the Home’: Deployable Ultra-Low Queuing Delay for All

Koen De Schepper† Olga Bondarenko* ‡ Ing-Jyh Tsang† Bob Briscoe‡
†Nokia Bell Labs, Belgium ‡Simula Research Laboratory, Norway
†{koen.de_schepper|ing-jyh.tsang}@nokia.com ‡{olgabo|bob}@simula.no

ABSTRACT

Traditionally, ultra-low queueing delay and capacity-seeking are considered mutually exclusive. We introduce an Internet service that offers both: Low Latency Low Loss Scalable throughput (L4S). Therefore it can incrementally replace best efforts as the default service. It uses ‘Scalable’ congestion controls, e.g. Data Centre TCP. Under a wide range of conditions emulated on a testbed using real residential broadband equipment, it proved hard not to get remarkably low (sub-millisecond) average queuing delay, zero congestion loss and full utilization. To realize these benefits we had to solve a hard problem: how to incrementally deploy controls like DCTCP on the public Internet. The solution uses two queues at the access link bottleneck, for Scalable and ‘Classic’ (Reno, Cubic, etc.) traffic. It is like a semi-permeable membrane, isolating their latency but coupling their capacity into a single resource pool. We implemented this ‘DualQ Coupled AQM’ as a Linux qdisc to test our claims. Although Scalable flows are much more aggressive than Classic, the AQM enables balance (TCP ‘fairness’) between them. However, it does not schedule flows, nor inspect deeper than IP. L4S packets are identified using the last ECN codepoint in the IP header, which the IETF is in the process of allocating.

CCS Concepts

•Networks → Cross-layer protocols; Packet scheduling; Network performance analysis; Public Internet; *Network resources allocation;*

*The first two authors contributed equally

ACM ISBN .
DOI:

Keywords

Internet, Performance, Queuing Delay, Latency, Scaling, Algorithms, Active Queue Management, AQM, Congestion Control, Congestion Avoidance, Congestion Signalling, Quality of Service, QoS, Incremental Deployment, TCP, Evaluation

1. INTRODUCTION

With increases in bandwidth, latency is becoming the critical performance factor for many, if not most, applications, e.g. Web, voice, conversational and interactive video, finance apps, online gaming, cloud-based apps, remote desktop. Latency is a multi-faceted problem that has to be tackled on many different fronts [9] and in all the different stages of application delivery—from data centres to access links and within end systems.

The aspect this paper addresses is the variable delay due to queuing. Even state-of-the art Active Queue Management (AQM) [38, 23] can only bring this down to roughly the same order as a typical base round-trip delay. This is because bottlenecks are typically in the most numerous edge access links where statistical flow multiplexing is lowest. And a single TCP flow will underutilize a link unless it can buffer about a round trip flight of data.

Queuing delay is intermittent, only occurring when a sufficiently long-running capacity-seeking flow (e.g. TCP) happens to coincide with interactive traffic [24]. However, intermittent delays dominate experience, and many real-time apps adapt their buffering to these intermittent episodes.

Our main contribution is to keep queueing delay extremely low (sub-millisecond) for *all* of a user’s Internet applications. A differentiated service (Diffserv) class such as EF [15] can provide low delay if limited to a fraction of the link’s traffic. Instead, we propose a new service that accommodates ‘greedy’ (capacity-seeking) applications that want both full link utilization and low queuing delay, so it can incrementally replace the default best efforts service. The new service effectively removes congestion loss as well, so it is called Low Latency, Low Loss, Scalable throughput (L4S).

L4S works because senders use one of the family of

'Scalable' congestion controls (§ 2.1 for the rationale). In contrast, we use the term 'Classic' for controls like TCP Reno and Cubic, where control becomes slacker as rate scales.

For evaluation we configure the host OS to use Data Centre TCP (DCTCP [1]), which is a widely available scalable control. We emphasize that the L4S service is not just intended for DCTCP, but also for a range of Scalable controls, e.g. Relentless TCP [34] and future scalable variants of QUIC, SCTP, real-time protocols, etc. In order to test one change at a time, we focus this paper on network-only changes, and use DCTCP, 'as is'. Our extensive experiments over a testbed using real data-centre and broadband access equipment and models of realistic traffic strengthen confidence that DCTCP would work very well over the public Internet.

However, DCTCP will need some safety (and performance) enhancements for production use, so a large group of DCTCP developers has informally agreed a list dubbed the 'TCP Prague' requirements (§ 5.2) to generalize from the otherwise confusing name.

Our second contribution is a solution to the deployability of Scalable controls like DCTCP. It is a common misconception that DCTCP is tailored for data centres, but the name merely emphasizes that it should not be deployed outside a controlled environment; it is too aggressive to coexist with existing 'Classic' traffic so a single admin is expected to upgrade all senders, receivers and bottlenecks at once.

We propose the 'Dual Queue Coupled AQM' that can be incrementally added at path bottlenecks to solve this 'coexistence' problem. It acts like a semi-permeable membrane. For delay it uses two queues to isolate L4S traffic from the Classic queue. But for throughput, the queues are coupled to appear as a single resource pool. So, for n aggressive L4S flows and m TCP-friendly Classic flows, each flow gets roughly $1/(n+m)$ of the capacity. The high-level idea of coupling is that the L4S queue emits congestion signals more aggressively to counterbalance the more aggressive response of L4S sources.

Balance between microflows should be a policy choice not a network default (§ 2.3), so we enable but do not enforce it. And coexistence between DCTCP and Classic flows is achieved without the network inspecting flows (no deeper than the IP layer). We have also tested that the L4S service can cope with a reasonable proportion of unresponsive traffic, just as best efforts copes with unresponsive streaming, VoIP, DNS etc.

The two queues are for transition, not scheduling priority. So low L4S delay is not at the expense of Classic performance and delay remains low even if a high load of solely L4S traffic fills the link.

Given access networks are invariably designed to bottleneck in one known location, the AQM does not have to be deployed in every buffer. Most of the benefit can be gained by deployment at the downstream queue into the access link, and home gateway deployment addresses the upstream. § 5 discusses how a Scalable control like DCTCP falls back to Reno if it encounters a non-L4S bottleneck. It also discusses wider deployment considerations, including other deployment scenarios such as coexistence between DCTCP and Classic TCP in heterogeneous or interconnected data centres.

L4S faces a very similar deployment problem to classic Explicit Congestion Notification (ECN [39]). However, we have learned from the ECN experience. To overcome the risk a first mover faces in kick-starting a multi-party deployment, we have attempted to ensure that the performance gain is dramatic enough to enable valuable new applications, not just a relatively marginal performance improvement.

The dramatic improvement of L4S has been demonstrated [7] by simultaneously running many apps that are both bandwidth-hungry and latency-sensitive over a regular 40Mb/s broadband access link. Two apps transmitted a user's physical movements (virtual reality goggles and pan/zoom finger gestures on a panoramic interactive video display) to cloud-based video servers over a broadband access (base delay 7 ms). The queuing delay of every packet was so low that the scenes that were generated on the fly and streamed back to the user seemed as if they were local and natural. Whereas without L4S, there was considerable lag and jerkiness. Other users were downloading streaming video, bulk files and running a gaming benchmark, all in the same queue, and mean per-packet queuing delay was around 500 μs.

Our third contribution is to ensure that the low queuing delay of L4S packets is preserved during overload from either L4S or Classic traffic, and neither can harm the other more than they would in a single queue.

Our fourth contribution is to ensure that the AQM can be deployed in any public Internet access network with zero configuration.

Our fifth contribution is extensive quantitative evaluation of the above claims: i) dramatically reduced delay and variability without increasing other impairments; ii) 'do no harm' to Classic traffic; iii) window balance between competing Scalable and Classic flows; and iv) overload handling (see § 4).

2. RATIONALE

2.1 Why a Scalable Congestion Control?

A congestion controller is defined as 'Scalable' if the rate of congestion signals per round trip, c, does not decrease as bandwidth-delay product (BDP or window) scales. The flow rate saw-tooths down whenever a congestion signal is emitted. So, by definition, the average sawtooth duration (a.k.a. recovery time) of a scalable control does not grow as rate scales.

TCP Cubic scales better than Reno, but it is still not fully scalable. For instance, for every 8-fold rate increase the average Cubic sawtooth duration doubles while its amplitude increases 8-fold, which is the cause ofgrowing delay variation. For instance, between 100 and 800 Mb/s, Cubic's sawtooth recovery time expands from 250 round trips to 500 round trips (assuming base RTT=20 ms). In contrast, whatever the rate, the average recovery time of a DCTCP sawtooth remains invariant at just half a round trip.

We use a Scalable congestion control because, unlike Classic TCP algorithms, this implies:

1. control does not slacken as the window scales;
2. variation of queuing and/or under-utilization, need not increase with scale (Figure 1).

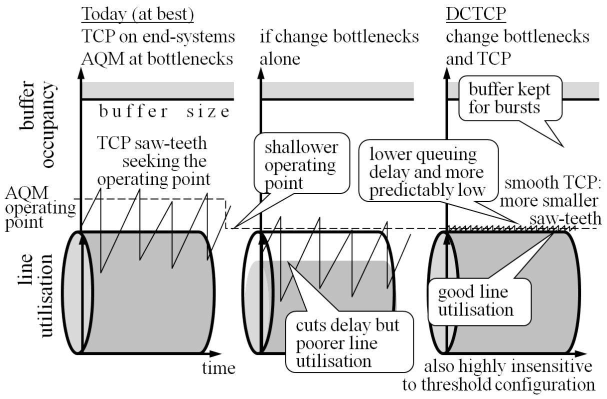


Figure 1: Data Centre TCP: Intuition

In the steady state, the number of signals per round is the product of segments per round W and the probability p that a segment carries a signal, i.e. $c = pW$. Formulas for the steady-state window, W, can be derived for each congestion controller (see § 3.1). Each formula is of the form $W \propto 1/p^B$, where B is a characteristic constant of the algorithm [4] (e.g. $B = 1/2$ for TCP Reno). So it is straightforward to state the above scalability condition in terms of B by substituting for p in the above formula for c:

$$c \propto W^{(1\ -\ 1/B)}.$$

Therefore, $B \geq 1$ defines a control as Scalable.

For DCTCP, $B \geq 1$, and DCTCP with probabilistic marking has $B = 1$ (see § 3.1) so the signalling rate is scale-invariant. DCTCP does not solve all scaling problems, e.g. its window update algorithm is unscalable by the definition in [29]. However, our AQM supports any scalable control, so we are confident that solutions to DCTCP's problems (e.g. [44]) will be able to evolve and co-exist with today's DCTCP, without a need for further network changes.

2.2 Why ECN?

Explicit Congestion Notification (ECN [39]) is purely a signal, whereas drop is both an impairment and a signal, which compromises signalling flexibility. ECN is essential to the L4S service, because:

1. A Scalable control's finer (more aggressive) saw-teeth imply a higher signalling rate, which would be untenable as loss, particularly during high load;
2. If the queue grows, a drop-based AQM holds back from introducing loss in case it is just a sub-RTT burst, whereas it can emit ECN immediately, because it is harmless.

This last point significantly reduces typical signalling delay, because with drop, the network has to add smoothing delay but it does not know each flow's RTT, so it has to smooth over a worst-case (inter-continental) RTT, to avoid instability for worst-case RTT flows. Whereas, the sender knows its own RTT, which it can use as the appropriate time constant to smooth the network's unsmoothed ECN signals [2] (and it can choose to respond without smoothing, e.g. in slow start).

Therefore, we require that Scalable traffic is ECN-capable, which we can also use to classify Scalable packets into the L4S queue (see § 3).

Irrespective of L4S, ECN also offers the obvious latency benefit of near-zero congestion loss, which is of most concern to short flows [40]. This removes retransmission and time-out delays and the head-of-line blocking that a loss can cause when a single TCP flow carries a multiplex of streams.

2.3 Why Not Per-Flow Queues?

Superficially, it might seem that per-flow queuing (as in FQ-CoDel) would fully address queuing delay; it is designed to isolate a latency-sensitive flow from the delays induced by other flows. However, that does not protect a latency-sensitive flow from the saw-toothing queue that a Classic TCP flow will still inflict upon *itself*. This is important for the growing trend of interactive video-based apps that are both extremely latency-sensitive and capacity-hungry, e.g. virtual and augmented reality, remote presence.

It might seem that self-inflicted queuing delay should not count. To avoid delay in a dedicated remote queue, a sender would have to hold back the data, causing the same delay, just in a different place. It seems preferable to release the data into a dedicated network queue; then it will be ready to go as soon as the queue drains.

However, this logic applies i) if and only if the sender somehow knows that the bottleneck in question implements per-flow queuing and ii) only for non-adaptive applications. Modern applications, e.g. HTTP/2 [5] or the panoramic interactive video app described in § 1, suppress lower priority data, depending on the progress of higher priority data sent already. To adapt how much they send, they need to maintain their self-induced send-queue locally, not remotely; because once optional data is in flight, they cannot suppress it.

As well as not solving self-induced latency, there are further well-rehearsed arguments against per-flow scheduling: i) it cannot know whether flow rate variations are deliberate, e.g. complex video activity ; ii) it cannot know (without prohibitive complexity) whether a flow using more, or less, than an equal share of a user's own capacity is intentional, or even mission-critical; iii) it needs to inspect transport layer headers (prevent-ing transport evolution); and iv) it requires many more queues and supporting scheduling structures.

Therefore we aim to reduce queuing delay without per-flow queuing. That does not preclude adding a per-flow policer, as a separate policy option.

3. SOLUTION DESIGN

The first design goal is ultra-low queuing delay for L4S traffic. However, if the number of flows at the bottleneck is small, Classic congestion controllers (CCs) need a significant queue to avoid under-utilization. One queue cannot satisfy two different delay goals so we classify any Classic traffic into a separate queue.

An L4S CC such as DCTCP achieves low latency, low loss and low rate variations by driving the network to give it frequent ECN marks. A Classic CC (TCP Reno, Cubic, etc.) would starve itself if confronted with such frequent signals.

So the second design goal is coexistence between Classic and L4S congestion controllers [26], meaning rough balance between their steady-state packet rates per RTT (a.k.a. TCP-fairness or TCP-friendliness). Therefore, we couple the congestion signals of the two queues and reduce the intensity for Classic traffic to compensate for its greater response to each signal, in a similar way to the single-queue coupled AQM in [16].

Packets are classified between the two queues based on the 2-bit ECN field in the IP header. Classic sources set the codepoints 'ECT(0)' or 'Not-ECT' depending on whether they do or do not support standard ('Classic') ECN [39]. L4S sources ensure their packets are classified into the L4S queue by setting 'ECT(1)', which is an experimental ECN codepoint being redefined for L4S (see § 5.1).

Introducing two queues creates a new problem: how often to schedule each queue. We do not want to schedule based on the number of flows in each, which would introduce all the problems of per-flow queuing (§ 2.3). Instead, we allow the end-systems to 'schedule' themselves in response to the congestion signals from each queue. However, whenever there is contention we give the L4S queue strict priority, because L4S sources can tightly control their own delay. Nonetheless, to prevent Classic starving, priority is conditional on the delay difference between the two queues. Also, if either or both queues become overloaded, low delay is preserved for L4S, but dropping behaves like a single queue AQM so that a misbehaving source can cause no more harm than in a single queue (see § 3.3).

The schematic in Figure 2 shows the whole DualQ Coupled AQM. with the classifier and scheduler as the first and last stages. In the middle, each queue has its own native AQM that determines dropping or marking even if the other queue is empty. The following subsections detail each part of the AQM, starting with the algorithm that couples the congestion signalling of L4S to that of the Classic AQM (for coexistence).

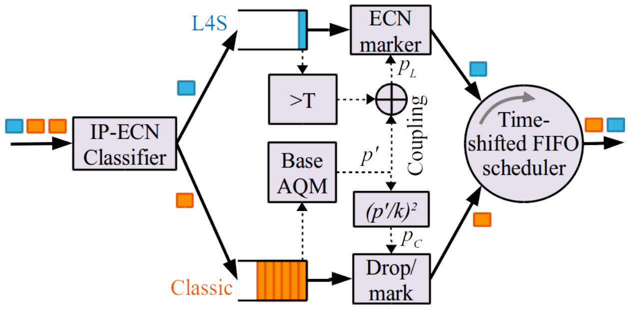


Figure 2: Dual Queue Coupled AQM

3.1 Coupled AQM for Window Balance

To support co-existence between the Classic (C) and L4S (L) congestion control families, we start with the equations that characterize the steady-state window, W, of each as a function of the loss or ECN-marking probability p. Then, like [16], we set the windows to be equal to derive the coupling relationship between the congestion signals for C and L.

We use Reno and DCTCP for C and L. We use Reno because it is the worst case (weakest). We can ignore dynamics, so we use the simplified Reno equation from [35]. For L4S, we do not use the equation from the DCTCP paper [1], which is only appropriate for step marking. Instead, we use the DCTCP equation that is appropriate to our coupled AQM, where marking is probabilistic, as derived in Appendix A of [16]. For balance between the windows, $W_{\text{reno}} = W_{\text{dc}}$, which becomes (1) by substituting from each window equation. Then we rearrange into a generalized relationship for coupling congestion signals in the network (2):

$$\sqrt{\frac{3}{2p_{\text{reno}}}} = \frac{2}{p_{\text{dc}}} \qquad (1) \qquad\qquad p_C = \left(\frac{p_L}{k}\right)^2, \qquad (2)$$

where coupling factor $k = 2\sqrt{2/3} = 1.64$ for Reno.

Appendix A of [16] shows that TCP Cubic [22] will be comfortably within its Reno compatibility mode for the 'Data Centre to the Home' scenarios that are the focus of this paper. The coupling formula in (2) also applies when the Classic traffic is TCP Cubic in Reno mode ('CReno'), except it should use $k = 2/1.68 = 1.19$.

To avoid floating point division in the kernel we round to $k = 2$. In all our experiments this proves to be a sufficiently accurate compromise for any Reno-friendly CC. It gives a slight window advantage to Reno, and a little more to CReno. However, any L4S source gives itself a counter-advantage by virtue of its shallower queue. So L4S achieves a higher packet rate with the same window because of it lower RTT. We do not expend effort countering this rate imbalance in the network—the proper place to address this is to ensure L4S sources will be less RTT-dependent (see § 5.2).

The coupling is implemented by structuring the AQM for Classic traffic in two stages (Figure 2). First what we call a 'Base AQM' outputs the internal probability

p'. This is used directly for L4S traffic ($p_L = p'$), but also transformed as per equation (2) to determine the dropping/marking probability for Classic packets (p_C).

Diversity of Base AQMs is possible and encouraged. Two have been implemented and tested [17]: a variant of RED and a proportional integral (PI) AQM. Both control queuing time not queue size, given the rate of each queue varies considerably [33, 37]. This paper uses the latter, because it performs better.

[16] also couples two AQMs to enable coexistence of different CCs, but within one queue, not across two. It proves theoretically and experimentally that a PI controller is a robust base AQM. It directly controls a scalable control like DCTCP (rate proportional to $1/p'$). And it shows that squaring the output of a PI controller is a more effective, more principled and simpler way of controlling TCP Reno (rate proportional to $1/\sqrt{p'}$) than PI Enhanced (PIE [38]). It shows that the piecewise lookup table of scaling values used by PIE was just a heuristic way of achieving the same effect as squaring.

3.2 Dual Queue for Low Latency

Often, there will only be traffic in one queue, so each queue needs its own native AQM. The L4S queue keeps delay low using a shallow marking threshold (T), which has already been proven for DCTCP. T is set in units of time [33, 3] with a floor of two packets, so it auto-tunes as the dequeue rate varies. On-off marking may [13] or may not [32, §5] be prone to instability. But to test one change at a time we deferred this to future research.

If there is traffic in both queues, an L4S packet can be marked either by its native AQM or by the coupled AQM (see the OR symbol in Figure 2). However, the coupling ensures that L4S traffic generally only touches the threshold when it is bursty or if there is insufficient Classic traffic.

Note that the L4S AQM emits ECN marks immediately and the sender is expected to do any necessary smoothing. Whereas Classic congestion signals are subject to smoothing delay in the network.

We use what we call a time-shifted FIFO scheduler [36] to decide between the head packets of the two queues. It selects the packet with the earliest arrival timestamp, after subtracting a constant timeshift to favour L4S packets. Normally, this behaves like a strict priority scheduler, but an L4S packet loses its priority if the extra delay of the leading Classic packet exceeds the timeshift. This protects Classic traffic from unresponsive L4S traffic or long L4S bursts, even ensuring a new Classic flow can break into a standing L4S queue.

3.3 Overload Handling

Having introduced a priority scheduler, during overload we must ensure it does no more harm to lower priority traffic than a single queue would.

Unresponsive traffic below the link rate just subtracts from the overall capacity, irrespective of whether it classifies itself as low (L4S) delay or regular (Classic) delay. Then the coupled AQM still enables other responsive flows to share out the remaining capacity by inducing the same balanced drop/mark probability as they would in a single queue with the same capacity subtracted.

To handle excessive unresponsive traffic, we simply switch the AQM over to using the Classic drop probability for both queues once the L4S marking probability saturates at 100%. By equation (2), if $k = 2$ this occurs once drop probability reaches $(100\%/k)^2 = 25\%$. When a DCTCP source detects a drop, it already falls back to classic behaviour, so balance between flow rates is preserved. The native L4S AQM also continues to ECN-mark packets whenever its queue exceeds the threshold, so any responsive L4S traffic maintains the ultra-low queuing delay of the L4S service.

If there are no packets in the Classic queue, the base AQM continues to evolve p' using the L4S queue. As soon as something starts to overload the L4S queue, this ensures the correct level of drop, given L4S sources fall back to a Classic response on detecting a drop. Nonetheless, with solely normal L4S sources, the L4S queue will stay shallow and drive the contribution from the base AQM (p') to zero.

3.4 Linux qdisc Implementation

Algorithm 1 Enqueue for Dual Queue Coupled AQM

```
STAMP(pkt)                                ▷ Attach arrival time to packet
if LQ.LEN() + CQ.LEN() >L then
    DROP(pkt)                             ▷ Drop packet if Q is full
else
    if LSB(ECN(pkt))==0 then              ▷ Not ECT or ECT(0)
        CQ.ENQUEUE(pkt)                   ▷ Classic
    else                                  ▷ ECT(1) or CE
        LQ.ENQUEUE(pkt)                   ▷ L4S
```

Algorithm 2 Dequeue for Dual Queue Coupled AQM

```
1:  while LQ.LEN() + CQ.LEN() >0 do
2:      if LQ.TIME() + D ≥ CQ.TIME() then
3:          LQ.DEQUEUE(pkt)                          ▷ L4S
4:          if (LQ.TIME() > T) ∨ (p > RAND()) then
5:              MARK(pkt)
6:      else
7:          CQ.DEQUEUE(pkt)                          ▷ Classic
8:          if  p > k * MAX(RAND(), RAND())  then
9:              if ECN(pkt)==0 then                  ▷ Not ECT
10:                 DROP(pkt)                        ▷ Squared drop
11:                 continue                         ▷ Redo loop
12:             else                                 ▷ ECT(0)
13:                 MARK(pkt)                        ▷ Squared mark
14:     RETURN(pkt)                ▷ return the packet, stop here
```

Algorithms 1 & 2 summarize the per packet enqueue and dequeue implementations of DualPI2 as pseudocode For clarity, overload and saturation logic are omitted. The full code is available as the Dualq option to the PI2 Linux qdisc implementation.[1] On enqueue, packets are time-stamped and classified. On dequeue, line 2 implements the time-shifted FIFO scheduler. It takes

[1]Open source at https://github.com/olgabo/dualpi2

the packet that waited the longest, after adding time-shift D to the L4S queuing time. If an L4S packet is scheduled, line 4 marks the packet either if the L4S threshold is exceeded, or if a random marking decision is drawn according to the probability p. If a Classic packet is scheduled, line 8 implements the squared probability p^2 without multiplication by dropping (or marking) the packet if both of two random comparisons are true. A useful *aide memoire* for this approach is "Think once to mark, twice to drop".

p is kept up to date by the core PI Algorithm (3) which only needs occasional execution [25]. The proportional gain factor β is multiplied by the change in queuing time. The integral gain factor α is typically smaller, to restore any persistent standing queue to the target delay. These factors, which can be negative, are added to the previous p every T_{update} (default 16 ms).

Algorithm 3 PI core: Every T_{update} p is updated

$curq = \text{CQ.TIME}()$
$p = p + \alpha * (curq - TARGET) + \beta * (curq - prevq)$
$prevq = curq$

4. EVALUATION

4.1 Testbed Setup

We used a testbed to evaluate the proposed DualQ AQM mechanism in a realistic setting, and to run repeatable experiments in a controlled environment. The testbed was assembled using carrier grade equipment in the same enviroment as for testing customer solutions. Figure 3 depicts the testbed, which consists of a classical residential service delivery network composed of Residential Gateway, xDSL DSLAM (DSL Access Multiplexer), BNG (Broadband Network Gateway), Service Routers (SR) and application servers. The Residential Gateway is connected by VDSL to a DSLAM, which is connected to the BNG through an aggregation network, representing a local ISP or access wholesaler. Traffic is routed to another network representing a global ISP that hosts the application servers and offers breakout to the Internet. The client computers in the home network and the application servers at the global ISP are Linux machines, which can be configured to use any TCP variant, start applications and test traffic. The two client-server pairs (A and B) are respectively configured with the same TCP variants and applications.

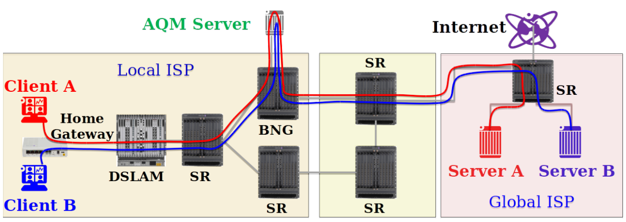


Figure 3: Testbed configuration

In a production access network, per-customer queues form the leaves of a hierarchical scheduling tree and they are deliberately arranged as the downstream bottleneck for each customer. Traffic from the client-server pairs is routed from the BNG through a Linux box ('AQM server'), which acts as the rate bottleneck where we configure the different AQMs being evaluated for the BNG. This server also emulates extra delay, controls the experiments, captures the traffic and analyses it. In practice it would also be important to deploy an AQM in the home gateway, but in our experiments the ACK traffic was below the upstream capacity.

The two client computers were connected to a modem using 100 Mbps Fast Ethernet; the xDSL line was configured at 48 Mbps downstream and 12 Mbps upstream; the links between network elements consisted of at least 1GigE connections. The base RTT (T_0) between the clients and servers was 7 ms, which was primarily due to the interleaved Forward Error Correction (FEC) configured for xDSL. We configured the different bottlenecks on the AQM server at the BNG on the downstream interface where the AQM was configured. Extra delay was configured on the upstream interface using a netem qdisc, to compose the total base RTTs tested.

To support higher bottleneck rates and lower RTTs all experiments were performed with the clients connected directly to the BNG with 1GigE connections. Those experiments fitting within xDSL limits were validated on the full testbed and compared, showing near identical results. All Linux computers were Ubuntu 14.04 LTS with kernel 3.18.9, which contained the implementations of the TCP variants and AQMs.

We used DCTCP for the Scalable congestion control and both Reno and Cubic for Classic, all with their default configurations[2]. In this paper we do not show Reno because the Cubic results were generally similar but not always as good. For ECN-Cubic, we enable TCP ECN negotiation. We compared DualPI2 with PIE and FQ-CoDel, all configured as in Table 1.

All	Buffer: 40000 pkt, ECN enabled
PIE	Target delay: 15 ms, Burst: 100 ms, TUpdate: 16 ms, α: 1/16, β: 10/16, ECN_drop: 25%
FQ-CoDel	Target delay: 5 ms, Burst: 100 ms
DualPI2	Target delay: 15 ms, L4S T: 1 ms, D: 30 ms, α: 5/16, β: 50/16, k: 2, ECN_drop: 100% L4S

Table 1: Default parameters for the different AQMs.

4.2 Experimental Approach

For traffic load we used long-running flows (§§ 4.3 & 4.4) and/or dynamic short flows (§ 4.5). We used long flows, not as an example of a realistic Internet traffic mix, rather to aid interpretation of various effects, such as starvation.

From all our experiments, we selected a representative subset to evaluate our two main performance

[2]Except DCTCP is patched to fix a bug that prevented it falling back to Reno on detecting a drop.

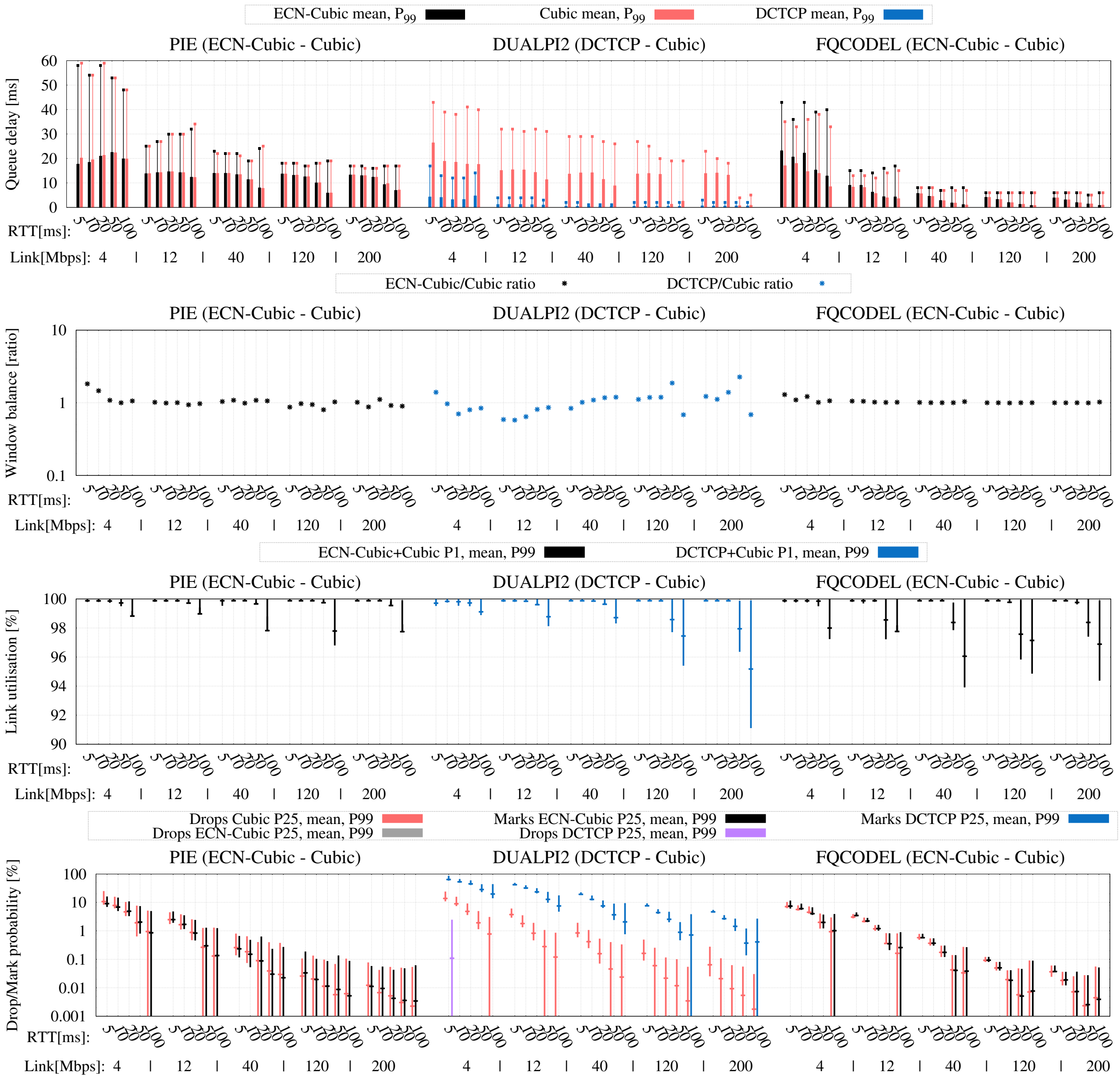


Figure 4: Equal RTT with 1 flow for each CC

goals: queuing delay and window balance. We also show rate balance, link utilization and drop/mark probability, as well as flow completion times in short flow experiments. Heavy load scenarios predominate in our selection, again not because they are typical, but because they do occur and they are the worst case.

We mixed different number of flows, evaluated flows with different congestion controls (CCs) and RTTs, and to verify behaviour on overload (§ 4.6), we injected unresponsive UDP load, both ECN and Not-ECN capable.

We configured PIE and FQ-CoDel with ECN as well as without, as a control so as not to attribute any performance gains to L4S ECN that are already available from Classic ECN. In this paper we present those combinations of CC and AQM that each AQM is intended to support: DCTCP with Cubic on DualPI2; and ECN-Cubic with Cubic on PIE and FQ-CoDel.

4.3 Experiments with long-running flows

Each experiment (lasting 250 s) was performed with a specified TCP variant configured on each client-server pair A and B and a specified AQM, bottleneck link speed and RTT on the AQM server. We performed a large number of experiments with different combinations of long-running flows, where each client started 0 to 10 file downloads on its matching server, resulting in 120 flow combinations competing at a common bottleneck for 250 seconds. These 120 experiments were executed for the 25 combinations of 5 RTTs (5, 10, 20, 50 and 100 ms) and 5 link speeds (4, 12, 40, 120 and

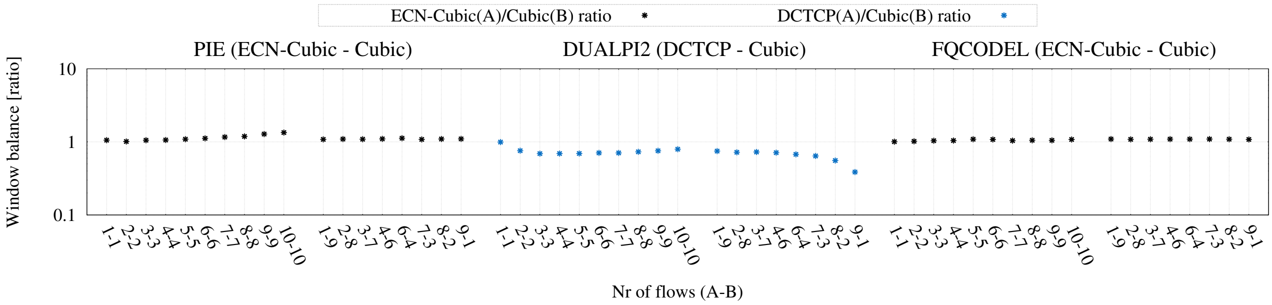


Figure 5: Different number of flows on a 40 Mbps link with 10 ms RTT.

200 Mbps).

For the 1-1 (one flow on pair A and one on B) combination Figure 4 shows queue delay, window ratio, link utilization and mark/drop probability for each AQM and congestion control. The results are plotted for different combinations of link speeds and RTTs on the x-axis.

Looking at queuing delay we can clearly see that L4S delay and delay variance are significantly lower than the other AQMs. All AQMs roughly hold to their target(s), except with higher delays for lower rates and some expected under-utilization for higher base RTTs. The lower link rates drive the non-ECN AQMs up to drop probabilities around 10%.

For the medium and high throughputs, L4S achieves sub-millisecond average delays with 99th percentile around 2–3 ms. The higher queuing delays for the smaller throughputs are due to the single packet serialization time of 3 ms (1 ms) on a 4 Mbps (12 Mbps) link. This is why we set a floor of 2 packets for the L4S marking threshold otherwise it would always mark 100%. The Cubic flows on the DualPI2 AQM achieve a similar average queuing delay as with the PIE AQM. Due to the time-shifted overload mechanism the 99th percentile of the Cubic flows pushes up the average and 99th percentile of the L4S queue delay.

The drop/mark plot clearly demonstrates the difference between drop and mark for the DualPI2 AQM. The squared drop probability results in near-equal windows for the different CCs, as demonstrated in the window balance plot. Due to the small queue delay of the L4S traffic, the total amount of packets in flight is smaller than with the other AQMs. To compensate, a higher drop and mark probability is needed. For the 4 Mbps and 5 ms base RTT, the probabilities sporadically start to exceed the coupled 25% drop and 100% mark thresholds, with some L4S drop as a result. For the higher BDPs, the links are less utilized due to the large window reduction of Cubic, resulting in more on/off-type marking for DCTCP. Even when DCTCP is not able to fill this gap due to its additive increase, it still reduces less than Cubic, with a higher DCTCP window as a result. For the very high BDPs Cubic starts to switch out of its Reno mode, resulting in the higher window of pure Cubic mode.

Figure 5 shows the window ratio for different combinations of numbers of long-running flow. The figure shows the results for a 40 Mbps link and 10 ms RTT, which was representative of the other link rates and RTTs. The number of flows for each pair (A and B) is shown on the x-axis: the first value is the total number of ECN-capable flows (ECN-Cubic or DCTCP), while the second is the number of Cubic flows.

The results show that in general window sizes are well-balanced with all combinations. This confirms that the simple squared coupling of the DUALPI2 AQM counterbalances the more aggressive response of DCTCP remarkably precisely over the whole range of combinations of flows.

Only when there are very few Classic flows compared to L4S flows does the DCTCP window become smaller. This is due to the low and bursty queue occupancy of Classic flows,which causes DCTCP flows to frequently hit the L4S threshold. This results in additional marking and a smaller window for DCTCP. A higher L4S-threshold removes this effect. As the higher throughput for one Classic flow is spread over multiple L4S flows, the throughput of the L4S flows is not heavily impacted, suggesting that if a compromise needs to be struck between low L4S latency and window balance, a low L4S threshold will always be preferable.

Throughput variance experiments with more than 2 flows (not shown due to space limitations) illustrate that, when a Classic flow competes with an L4S, it conveys its variations to the L4S flow (which fills up the gaps). However, when solely DCTCP flows compete their rates are much more stable.

4.4 Experiments with different RTTs

To evaluate the RTT-dependence of the windows and rates of different CCs, we conducted additional experiments with one flow per client server pair, each having different base RTTs. These experiments were repeated for the 5 link speeds.

Figure 6 shows queue delay and window and rate ratio for flows with unequal RTTs, running concurrently. We use one flow for each congestion control, labelled as flow A for ECN congestion controls (ECN-Cubic or DCTCP) and flow B for Cubic. Different combinations of RTTs for each of the flows are shown on the x-axis.

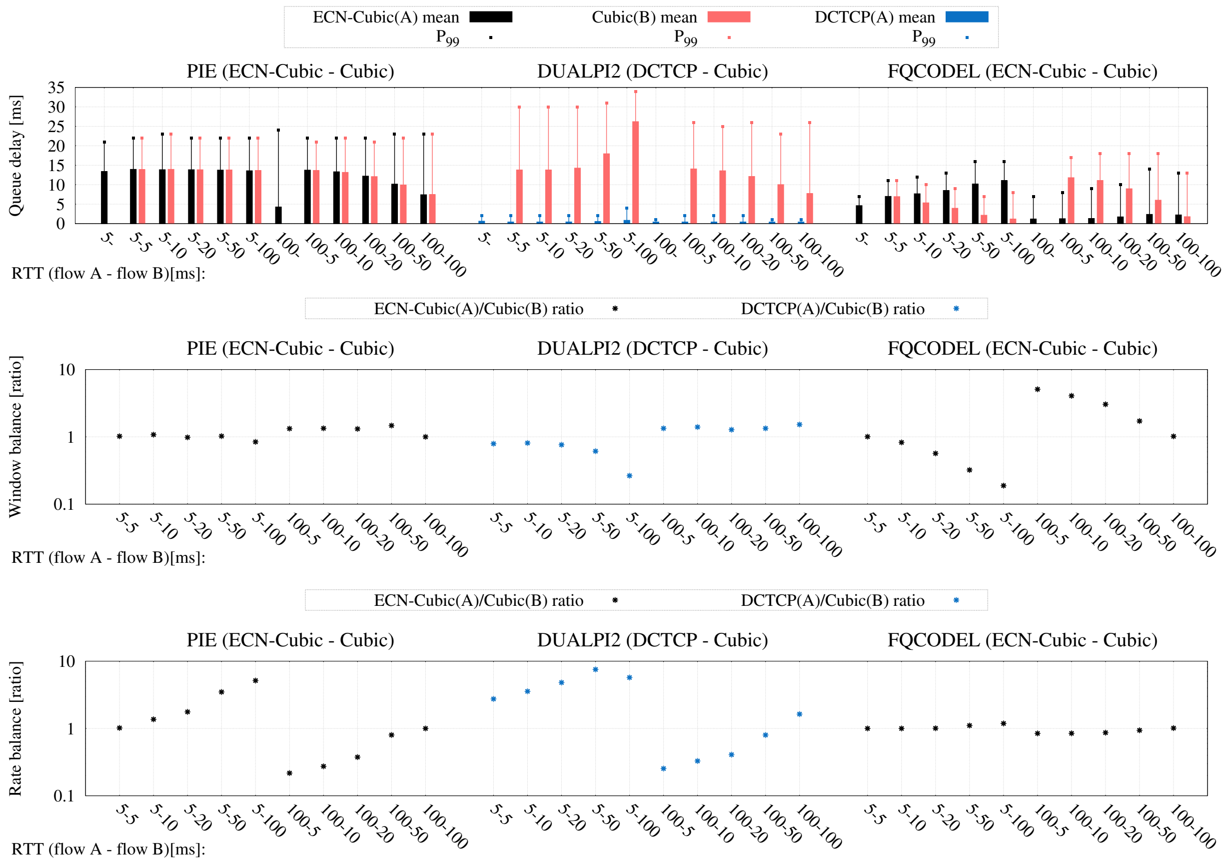


Figure 6: Mixed RTT with 1 flow for each CC on a 40 Mbps link.

For example, 5-20 means 5 ms base RTT for flow A and 20 ms for flow B.

Looking first at queuing delays in the DualPI2 AQM, it can be seen that the extremely low latency for L4S traffic is preserved in all cases, including in the presence of longer RTT traffic. Large-RTT Classic flows combined with small-RTT L4S flows result in a longer average Classic queue (see A-B = 5-100). This is again due to the bursty character of ACK-clocked Classic TCP flows, which need to wait until the L4S traffic has backed off sufficiently to create scheduling opportunities for the Classic flows. This effect is tempered by the time-shifted scheduler, which limits the waiting time for the burst to 30 ms at the expense of higher 99$^{\text{th}}$ percentile delay for the L4S traffic.

In this same 5-100 case, window balance also suffers. The bursty Classic traffic with its associated higher L4S threshold marking drags down the L4S window size.

Comparing the bottom two plots, particularly in the 5-100 case, with PIE or DUALPI2 it can be seen that window balance leads to considerable rate imbalance. This is not surprising, because it is well known that competing TCP flows equalize their congestion windows so their bit rates will be inversely proportional to their RTTs. However, as AQM reduce queueing delay they intensify this effect, because the ratio between total RTTs tends towards the ratio between base RTTs. The implications of this trend are discussed in § 5.2.

For instance, in the 5-100 case when the ratio between base RTTs is 20×, the ratio between flow rates is about 6×. This is because PIE holds queuing delay at about 15 ms, and $(100 + 15)/(5 + 15) \approx 6$.

L4S all-but eliminates queuing delay so total RTT is hardly any greater than base RTT. Therefore even for the 5-50 case, rate imbalance is already approaching 10×. In the 5-100 case, it can be seen that the rate-imbalance trend reverses. However, this is due to the increased variance of the L4S queue in response to increased Cubic burstiness as discussed above. In other experiments (not shown) with the burstiness of Cubic removed by using 2 DCTCP flows alone, rate imbalance does indeed tend towards the inverse of the ratio between the base RTTs of the flows.

Conversely, with FQ-CoDel the Flow Queuing scheduler enforces rate balance, which necessarily requires considerable window imbalance.

4.5 Experiments with dynamic short flows

On top of the long flow experiments, we added emulated web traffic load patterns between each client-server pair, to evaluate the dynamic behaviour of the AQMs with their congestion controllers. For this we

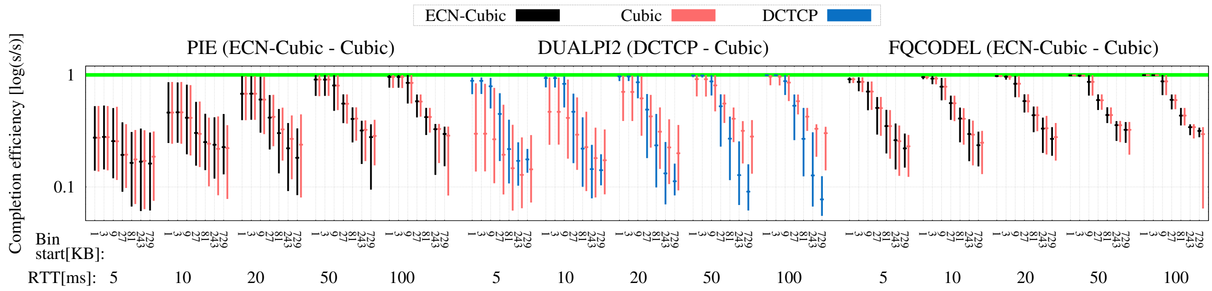


Figure 8: Heavy dynamic workload: 1 long flow and 300 short requests per second for each CC on a 120 Mbps link with equal 10 ms base RTT. The bin boundaries are 1 KB, 3 KB, 9 KB, 27 KB, 81 KB, 243 KB, 729 KB and 1 MB.

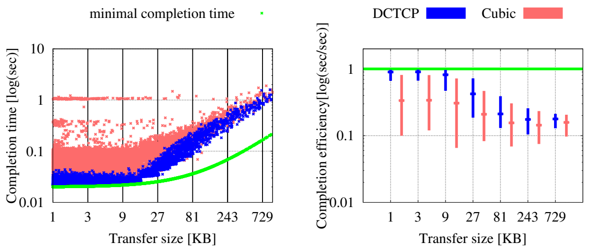


Figure 7: Completion time against efficiency representation for 1 long flow and high dynamic load each on a 40 Mbps link with 10 ms base RTT.

used an exponential arrival process with an average of 1 (low load) or 10 (high load) requested items per second for the 4 Mbps link capacity, scaled for the higher link speeds up to 50 (low) or 500 (high) requests for the 200 Mbps links. Every request opened a new TCP connection, closed by the server after sending data with a size according to a Pareto distribution with $\alpha = 0.9$ and a minimum size of 1 KB and maximum 1 MB. The client logged the completion time and downloaded size. Timing was started just before opening the TCP socket, and stopped after the close by the server was detected.

The left-hand side of Figure 7 shows a log-log scatter plot of the completion time to item size relation for the high load DualPI2 AQM test case on a 40 Mbps link with 10 ms base RTT. The green line is the theoretically achievable completion time, taking the RTT into account but downloading at full link speed from the start. As can be seen, the L4S short flows (within the initial window size of 10 packets) closely achieve this. They leave the TCP stack in a burst and face very low delay in the network. This same representation also helps in understanding where Classic download time is typically lost. Around 1 second a lot of downloads had to wait for the retransmission time-out after lost SYN packets. Around 200 ms the minimum retransmission time-out for tail loss is clearly visible. Long flows share the throughput better, which is why they are further from the theoretical completion time for a lone flow.

To better quantify the average and percentiles of the completion times, we used the Completion Efficiency representation on the right of Figure 7. To calculate its completion efficiency for each item we divided actual by theoretical completion time. We then binned the samples in log scale bins (base 3) and calculated the average, 1^{st} and 99^{th} percentiles. The green theoretical completion time is now at 1 (maximum efficiency).

Figure 8 shows completion efficiency for a high load of short flows plus a single long-running flow for each congestion control on a 120 Mbps link with different RTTs.

With DualQ or FQ the completion times of short flows are near-ideal. DualQ achieves this by keeping the queue very shallow for all L4S flows. In contrast FQ explicitly identifies and priority-schedules short flows.

In higher BDP cases, and in the high load case shown, the completion times of larger downloads are longer with DualPI2 than with the other AQMs. This is partly due to the additional marking of bursty traffic due to the shallow L4S threshold, which gives Cubic flows an advantage (as already discussed). However, the primary cause is a known problem with DCTCP convergence time. When a DCTCP flow is trying to push in against a high load of other DCTCP flows, it drops out of slow start very early, because of the higher prevailing marking level. Then it falls back to pushing in very slowly using only additive increase. Similarly, when another flow departs, the additive increase of DCTCP takes many round trips to fill the newly available capacity.

Others have noticed this problem and modified the additive increase of DCTCP [44]. Nonetheless, DCTCP slow start also has to be modified—the aggression of slow start in one flow has to increase to match the increased aggression of congestion avoidance in others. Solving this problem is included in the TCP Prague requirements (see § 5.2), but it is outside the AQM-only scope of the present paper.

Figure 9 adds further weight to the argument that DCTCP, not the DualQ AQM, is the cause of the longer completion times. Average queue delay, queue variance and link utilization are all better with L4S/DualQ than with FQ-CoDel. So it seems that DCTCP is just not exploiting these advantages.

If we now compare the results in Figure 9 with those

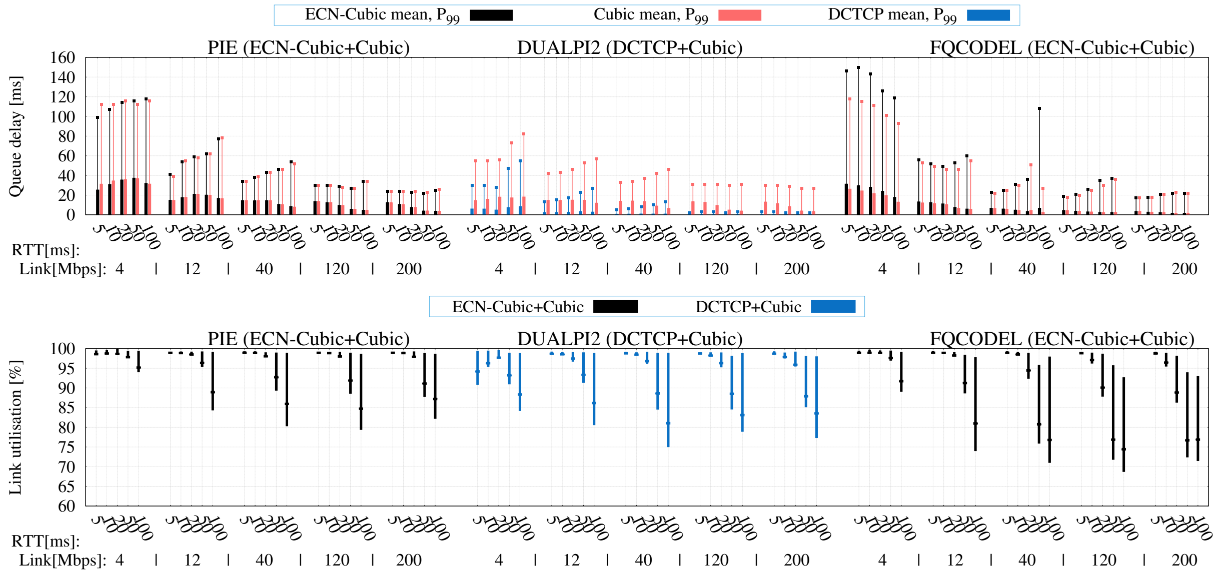


Figure 9: Heavy dynamic workload: 1 long flow and 300 short requests per second for each CC.

for just long-running flows in Figure 4, we see the effect of adding dynamic flows. They dramatically increase queue delay variance (note the change in scale), particularly with FQCODEL and PIE. Nonetheless, L4S queuing delay is still extremely low, with only a slight increase in variance.

Comparing the link utilization plots, the added dynamic flows universally reduce utilization as arriving flows take a while to use up the capacity that departing flows vacate, particularly at higher RTTs. With DUALPI2, under-utilization is only a little worse than with PIE, despite DCTCP's convergence problem (discussed above). This is because the Classic Cubic traffic takes up some of the slack.

4.6 Overload experiments

To validate the correct overload behaviour, we added an unresponsive UDP flow to 5 long-running flows of each congestion control type (ECN and non-ECN) over a 100 Mbps bottleneck link with 10 ms base RTT. For each AQM we ran 2 sets of tests with the UDP traffic marked as either ECN/L4S or non-ECN. Each set tested 5 different UDP rates (50, 70, 100, 140 and 200 Mbps).

Figure 10 shows the results for the DualQ AQM. The top plot shows the link output rate for each traffic type. The more the UDP flow squeezes the responsive flows, the more they drive up the congestion level (ECN or drop). Only responsive flows heed ECN marks. So, in the ECN UDP flow case, before congestion reaches the level where the AQM starts dropping ECN packets, the UDP flow is unaffected by congestion.

Once the AQM starts dropping ECN packets (and in the non-ECN UDP flow case), the drop probability necessary to make the responsive flows fit into the remaining capacity also subtracts from the UDP flow, freeing up some extra capacity for the responsive traffic.

The capacity left by the UDP flow for responsive traffic is roughly the same whether the UDP flow uses L4S-ECN or not, but the largest difference is where the arrival rate of the UDP flow is around 100% of the capacity. Once unresponsive traffic significantly exceeds 100%, it leaves very little capacity for the responsive traffic.

All this behaviour was exactly the same as with a single queue AQM (i.e. PIE), which was our intention. We wanted to ensure that introducing two queues would not introduce any new pathologies. Then any applications relying on unresponsive behaviour should work the same, and any optional mechanisms to police unresponsive flows should also work the same.

In contrast, flow queuing starts dropping unresponsive traffic when it exceeds an equal share of throughput. For instance, a 50 Mbps flow experiences about 80% drop, to force it to share the capacity equally with 10 other flows.

The middle plot shows that the windows of the DCTCP and Cubic flows balance as long as the unresponsive traffic is no greater than the link capacity. For higher levels of unresponsive traffic, the throughput of the responsive traffic is more dominated by long retransmission time-outs, which results in more equal rates, causing window imbalance because of the different RTTs.

Finally the bottom plot shows the queuing delay for the DualQ during the same experiments. The most notable feature is that, whether the unresponsive traffic is L4S or Not ECN, average L4S queuing delay remains below about 2 ms, except in the L4S UDP case, and then only once it exceeds 140% of capacity.

In the case when the UDP traffic is not ECN, the PI2

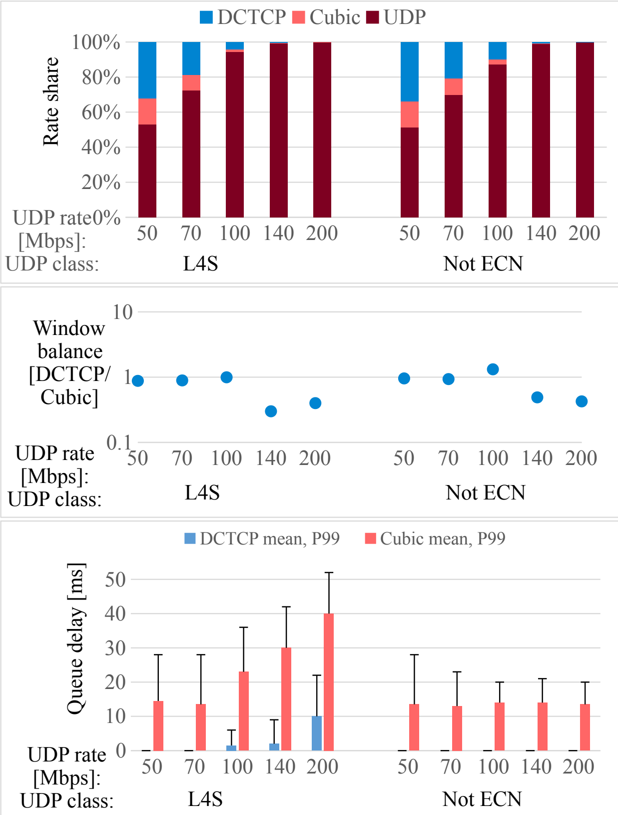


Figure 10: Overload experiments on a 100 Mbps link

AQM holds Classic queue delay to its target by applying sufficient drop. The coupled AQM translates this to a high level of L4S marking or, if congestion is high, it applies the same level of drop to both queues. Given L4S throughput is relatively low in this case, it is easy for L4S queuing delay to remain very low.

In the case when the UDP traffic is L4S, the majority of the load arrives at the L4S queue. The native L4S threshold only applies marking, which the UDP traffic ignores. So the overload mechanism described in § 3.3 starts to dominate. This takes over whenever the Classic queue is empty, which happens increasingly often as more UDP L4S traffic arrives. At such times, the base AQM (PI controller) uses the L4S queue delay to drive its output, still aiming for the Classic 15 ms target. The more unresponsive traffic that arrives at the L4S queue, the more the L4S queue shifts from the 1 ms L4S threshold to the 15 ms Classic target. This effect can be seen between 100% and 200% in the L4S UDP case.

5. DEPLOYMENT CONSIDERATIONS

5.1 Standardization Requirements

The IETF has taken on L4S standardization work, in principle. It has adopted a proposal [6] to make the ECT(1) codepoint available for experimental classification of L4S packets at the IP layer (v4 and v6), as described in § 3. [18] considers the pros and cons of various candidate identifiers and finds that none are without problems, but proposes ECT(1) as the least worst.

The main issue is that there is only one spare codepoint, so a queue can distinguish L and C packets, but congestion marking has to use the same Congestion Experienced (CE) codepoint for both L & C packets. This is not a problem for hosts but, in the (unusual) case of multiple simultaneous bottlenecks, any packet already marked CE upstream will have to be classified into the L queue, irrespective of whether it was originally C or L. This is considered acceptable for TCP given that, if a few packets arrive early out of order, subsequent packets still advance the ACK counter.

Operators will be able to classify L4S on additional identifiers (e.g. by ECN plus address range or VLAN ID), which they might use for initial exclusivity, without compromising long-term interoperability.

The IETF also plans to define the semantics of the new identifier. The 'Classic' ECN standard [39] defines a CE mark as equivalent to a drop, so queuing delay with Classic ECN cannot be better than with drop (this may be why operators have not deployed Classic ECN [41, § 5]). The square relationship between an L4S mark and a drop in this paper (Eqn. (2)) has been proposed for experimental standardization [18]. Nonetheless, it has been proposed to recommend rather than standardize a value for the coupling factor, k, given differences would not prevent interoperability.

The IETF is also adopting a specification of the dualQ coupled AQM mechanism [17] so that multiple implementations can be built, tested and compared, possibly using different base AQMs internally.

5.2 Congestion Control Roadmap

This paper uses DCTCP unmodified[3] in all experiments i) to focus the parameter space of our experiments on the network mechanism, without which end-system performance improvements would be moot; and ii) to emphasize that the end-system side of the multi-party deployment is already available (in the Linux mainline and Windows), at least for testing purposes. Nonetheless, numerous improvements to DCTCP can be envisaged for this new public Internet scenario. They are listed below in priority order starting with those necessary for safety, and ending with performance improvements. They are adapted from the congestion control requirements identified in the IETF L4S architecture draft [12], which are in turn adapted from the "TCP Prague requirements", named after the meeting in Prague of a large group of DCTCP developers that informally agreed them [8]:

1. Fall back to Reno/Cubic on loss (Windows does, but Linux does not due to a bug—fix submitted);
2. Negotiate altered feedback semantics [30, 11];
3. Use of a standardized packet identifier [18];

[3]See footnote 2.

4. Handle a window of less than 2, rather than grow the queue if base RTT is low [10];
5. Smooth ECN feedback over a flow's own RTT, not the RTT hard-coded for data-centres [2, § 5];
6. Fall back to Reno/Cubic if increased delay of classic ECN bottleneck detected;
7. Faster-than-additive increase, e.g. Adaptive Acceleration (A^2DTCP) [44];
8. Less drastic exit from slow-start, similar goal to Flow-Aware (FA-DCTCP) [27];
9. Reduce RTT-dependence of rate [2, § 5] (see below).

With tail-drop queues, so-called 'RTT-unfairness' had never been a great cause for concern because the RTTs of all long-running flows included a common queuing delay component that was no less than worst-case base RTT (due to the historical rule of thumb for sizing access link buffers[4] at 1 worst-case RTT). So, even where the ratio between base delays was extreme, the ratio between total RTTs rarely exceeded 2 (e.g. if worst-case base RTT is 100 ms, worst-case total RTT imbalance tends to $(100 + 100)/(0 + 100)$.

However, Classic AQMs reduce queuing delay to a typical, rather than worst-case, RTT. For instance, with PIE, the queuing delay common to each flow is 15 ms. Therefore, worst-case rate imbalance will be $(100 + 15)/(0 + 15) \approx 8$ (see the explanation in § 4.4 of the rate imbalance in Figure 6).

Because of the cushioning effect of queuing delay, even when base RTTs are extremely imbalanced rates are not. But, because L4S all-but eliminates queuing delay, it exposes the full effect of the 'RTT-unfairness' issue.

We do not believe the network needs to be involved in addressing this problem. RTT-dependence is a feature of end-to-end congestion controls, so that is where it should be addressed. Classic CCs will not need to change, because classic queues will still need to be large to avoid under-utilization. However, L4S congestion controls will need to be less RTT-dependent, to avoid starving any L4S and Classic flows with larger RTTs (hence reduced RTT-dependence has been added to the TCP-Prague requirements above).

As a fortunate side-effect, it will be easier to define the coupling factor k (see § 3.1) to balance throughput between RTT-independent L4S traffic and large-queued Classic traffic.

5.3 Deployment Scenarios

The applicability of the DualQ is of course not limited to fixed public access networks. The DualQ Coupled AQM should also enable DCTCP to be deployed across multi-tenant data centres or across community of interest networks connecting private data centres—anywhere where the lack of a centralized system-admin makes coordinated deployment of DCTCP impractical. The most likely DC bottlenecks could be prioritized for deployment, e.g. at the ingress and egress of hypervisors or top-of-rack switches depending on topology, and at WAN access points.

In mobile networks the bottleneck is usually the radio access where buffering is more complex, but in principle an AQM similar to the Coupled DualQ ought to work.

6. RELATED WORK

In 2002, Gibbens and Kelly [21] developed a scheme to mark ECN in a priority queue based on the combined length of both queues. However, they were not trying to serve different congestion controllers as in the present work. In 2005 Kuzmanovic [32, §5] presaged the main elements of DCTCP showing that ECN should enable a naïve unsmoothed threshold marking scheme to outperform sophisticated AQMs like the proportional integral (PI) controller. It assumed smoothing at the sender, as earlier proposed by Floyd [19].

Wu et al. [42] investigates a way to incrementally deploy DCTCP within data centres, marking ECN when the temporal queue exceeds a shallow threshold but using standard ECN [39] on end-systems. Kuhlewind et al. [31] showed that DCTCP and Reno could co-exist in the same queue configured with a form of WRED [14] classifying on ECN not Diffserv. Judd [28] uses Diffserv scheduling to partition data centre switches between DCTCP and classic traffic in a financial data centre scenario, but as already explained this relies on management configuration based on prediction of the traffic matrix and its dynamics, which becomes hard on low stat-mux links. Fair Low Latency (FaLL) [43] is an AQM for DC switches building on CoDel [37]. Unlike the DualQ, FaLL inspects the transport layer of sample packets to focus more marking onto faster flows while keeping the queue short.

7. CONCLUSION

Classic TCP induces two impairments: queuing delay and loss. A good AQM can reduce queuing delay but then TCP induces higher loss. In a low stat-mux link, there is a limit to how much an AQM can reduce queuing delay without TCP's sawteeth introducing a third impairment: under-utilization. Thus TCP is like a balloon: when the network squeezes one impairment, another bulges out.

This paper moves on from debating where the network should best squeeze the TCP balloon. It recognizes that the problem is now wholly outside the network: Classic TCP (the balloon itself) is the problem. But this does not mean the solution is also wholly outside the network. This paper has shown that the network plays a crucial role in enabling hosts to transition away from the Classic TCP balloon. The 'DualQ Coupled AQM' detailed in this paper is not notable as somehow a 'better' AQM than others. Rather, it is notable

[4]Note that access buffers cannot exploit such high flow aggregation as in the core [20]

as a coupling between two AQMs in two queues—as a transition mechanism to enable hosts to kick out their old TCP balloon.

Hosts will then be able to transition to a member of the family of scalable congestion controls. This can still be likened to a balloon. But it is a tiny balloon (near-zero impairments) and, importantly, it will stay the same tiny size (invariant impairments as BDP scales). Whereas the Classic TCP balloon is continuing to grow (worsening impairments) as BDP scales. This is why we call the new Internet service 'Low Latency Low Loss Scalable throughput' (L4S).

The paper provides not just the mechanism but also the incentive for transition—the tiny size of all the impairments. For link rates from 4–200 Mb/s and RTTs from 5–100 ms, our extensive testbed experiments with a wide range of heavy load scenarios have shown near-zero congestion loss; sub-millisecond average queuing delay (roughly 500 μs) with tight variance; and near-full utilization.

We have been careful as far as possible to do no harm to those still using the Classic service. Also, given the network splits traffic into two queues, when it merges them back together, we have taken great care that it does not enforce flow 'fairness'. Nonetheless, if hosts are aiming for flow 'fairness' they will get it, while remaining oblivious to the difference between Scalable and Classic congestion controls.

We have been careful to handle overload in the same principled way as normal operation, preserving the same ultra-low delay for L4S packets, and dropping excess load as if the two queues were one.

And finally, we have been careful to heed the zero-config requirement of recent AQM research, not only ensuring the AQMs inherently auto-tune to link rate, but also shifting RTT-dependent smoothing to end-systems, which know their own RTT.

8. REFERENCES

[1] Alizadeh, M., Greenberg, A., Maltz, D. A., Padhye, J., Patel, P., Prabhakar, B., Sengupta, S., and Sridharan, M. Data Center TCP (DCTCP). *Proc. ACM SIGCOMM'10, Computer Communication Review 40*, 4 (Oct. 2010), 63–74.

[2] Alizadeh, M., Javanmard, A., and Prabhakar, B. Analysis of DCTCP: Stability, Convergence, and Fairness. *Proc. ACM SIGMETRICS'11* (2011).

[3] Bai, W., Chen, K., Chen, L., Kim, C., and Wu, H. Enabling ECN over Generic Packet Scheduling. In *Proc. Int'l Conf Emerging Networking EXperiments and Technologies* (New York, NY, USA, 2016), CoNEXT '16, ACM, pp. 191–204.

[4] Bansal, D., and Balakrishnan, H. Binomial Congestion Control Algorithms. In *Proc. IEEE Conference on Computer Communications (Infocom'01)* (Apr. 2001), IEEE, pp. 631–640.

[5] Belshe, M., Peon, R., and Thomson (Ed.), M. Hypertext Transfer Protocol version 2 (HTTP/2). Request for Comments 7540, RFC Editor, May 2015.

[6] Black, D. Explicit Congestion Notification (ECN) Experimentation. Internet Draft draft-ietf-tsvwg-ecn-experimentation-00, Internet Engineering Task Force, Dec. 2016. (Work in Progress).

[7] Bondarenko, O., De Schepper, K., Tsang, I.-J., Briscoe, B., Petlund, A., and Griwodz, C. Ultra-Low Delay for All: Live Experience, Live Analysis. In *Proc. ACM Multimedia Systems; Demo Session* (New York, NY, USA, May 2016), ACM, pp. 33:1–33:4.

[8] Briscoe, B. [tcpPrague] Notes: DCTCP evolution 'bar BoF': Tue 21 Jul 2015, 17:40, Prague. Archived mailing list posting URL: https://mailarchive.ietf.org/arch/msg/tcpprague/mwWncQg3egPd15FItYWiEvRDrvA, July 2015.

[9] Briscoe, B., Brunstrom, A., Petlund, A., Hayes, D., Ros, D., Tsang, I.-J., Gjessing, S., Fairhurst, G., Griwodz, C., and Welzl, M. Reducing Internet Latency: A Survey of Techniques and their Merits. *IEEE Communications Surveys & Tutorials 18*, 3 (Q3 2016), 2149–2196.

[10] Briscoe, B., and de Schepper, K. Scaling TCP's Congestion Window for Small Round Trip Times. Technical report TR-TUB8-2015-002, BT, May 2015. http://riteproject.eu/publications/.

[11] Briscoe, B., Kühlewind, M., and Scheffenegger, R. More Accurate ECN Feedback in TCP. Internet Draft draft-ietf-tcpm-accurate-ecn-02, Internet Engineering Task Force, Oct. 2016. (Work in Progress).

[12] Briscoe (Ed.), B., De Schepper, K., and Bagnulo, M. Low Latency, Low Loss, Scalable Throughput (L4S) Internet Service: Architecture. Internet Draft draft-briscoe-tsvwg-l4s-arch-00, Internet Engineering Task Force, Oct. 2016. (Work in Progress).

[13] Chen, W., Cheng, P., Ren, F., Shu, R., and Lin, C. Ease the Queue Oscillation: Analysis and Enhancement of DCTCP. In *Distributed Computing Systems (ICDCS), 2013 IEEE 33rd International Conference on* (July 2013), pp. 450–459.

[14] Clark, D. D., and Fang, W. Explicit allocation of best-effort packet delivery service. *IEEE/ACM Transactions on Networking 6*, 4 (Aug. 1998), 362–373.

[15] Davie, B., et al. An Expedited Forwarding PHB (Per-Hop Behavior). Request for Comments

3246, Internet Engineering Task Force, Mar. 2002.

[16] De Schepper, K., Bondarenko, O., Tsang, I.-J., and Briscoe, B. PI^2 : A Linearized AQM for both Classic and Scalable TCP. In *Proc. ACM CoNEXT 2016* (New York, NY, USA, Dec. 2016), ACM.

[17] De Schepper, K., Briscoe (Ed.), B., Bondarenko, O., and Tsang, I.-J. DualQ Coupled AQM for Low Latency, Low Loss and Scalable Throughput. Internet Draft draft-briscoe-tsvwg-aqm-dualq-coupled-00, Internet Engineering Task Force, Oct. 2016. (Work in Progress).

[18] De Schepper, K., Briscoe (Ed.), B., and Tsang, I.-J. Identifying Modified Explicit Congestion Notification (ECN) Semantics for Ultra-Low Queuing Delay. Internet Draft draft-briscoe-tsvwg-ecn-l4s-id-02, Internet Engineering Task Force, Oct. 2016. (Work in Progress).

[19] Floyd, S. TCP and Explicit Congestion Notification. *ACM SIGCOMM Computer Communication Review 24*, 5 (Oct. 1994), 10–23. (This issue of CCR incorrectly has '1995' on the cover).

[20] Ganjali, Y., and McKeown, N. Update on Buffer Sizing in Internet Routers. *ACM SIGCOMM Computer Communication Review 36* (Oct. 2006).

[21] Gibbens, R. J., and Kelly, F. P. On Packet Marking at Priority Queues. *IEEE Transactions on Automatic Control 47*, 6 (June 2002), 1016–1020.

[22] Ha, S., Rhee, I., and Xu, L. CUBIC: a new TCP-friendly high-speed TCP variant. *SIGOPS Operating Systems Review 42*, 5 (July 2008), 64–74.

[23] Hoeiland-Joergensen, T., McKenney, P., Täht, D., Gettys, J., and Dumazet, E. The FlowQueue-CoDel Packet Scheduler and Active Queue Management Algorithm. Internet Draft draft-ietf-aqm-fq-codel-06, Internet Engineering Task Force, Mar. 2016. (work in progress).

[24] Hohlfeld, O., Pujol, E., Ciucu, F., Feldmann, A., and Barford, P. A QoE Perspective on Sizing Network Buffers. In *Proc. Internet Measurement Conf (IMC'14)* (Nov. 2014), ACM, pp. 333–346.

[25] Hollot, C. V., Misra, V., Towsley, D., and Gong, W. Analysis and design of controllers for AQM routers supporting TCP flows. *IEEE Transactions on Automatic Control 47*, 6 (Jun 2002), 945–959.

[26] Irteza, S., Ahmed, A., Farrukh, S., Memon, B., and Qazi, I. On the Coexistence of Transport Protocols in Data Centers. In *Proc. IEEE Int'l Conf. on Communications (ICC 2014)* (June 2014), pp. 3203–3208.

[27] Joy, S., and Nayak, A. Improving Flow Completion Time for Short Flows in Datacenter Networks. In *Int'l Symposium on Integrated Network Management (IM 2015)* (May 2015), IFIP/IEEE, pp. 700–705.

[28] Judd, G. Attaining the Promise and Avoiding the Pitfalls of TCP in the Datacenter. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)* (Oakland, CA, May 2015), USENIX Association, pp. 145–157.

[29] Kelly, T. Scalable tcp: Improving performance in highspeed wide area networks. *ACM SIGCOMM Computer Communication Review 32*, 2 (Apr. 2003).

[30] Kühlewind, M., Scheffenegger, R., and Briscoe, B. Problem Statement and Requirements for Increased Accuracy in Explicit Congestion Notification (ECN) Feedback. Request for Comments 7560, RFC Editor, Aug. 2015.

[31] Kühlewind, M., Wagner, D. P., Espinosa, J. M. R., and Briscoe, B. Using Data Center TCP (DCTCP) in the Internet. In *Proc. Third IEEE Globecom Workshop on Telecommunications Standards: From Research to Standards* (Dec. 2014), pp. 583–588.

[32] Kuzmanovic, A. The Power of Explicit Congestion Notification. *Proc. ACM SIGCOMM'05, Computer Communication Review 35*, 4 (2005).

[33] Kwon, M., and Fahmy, S. A Comparison of Load-based and Queue-based Active Queue Management Algorithms. In *Proc. Int'l Soc. for Optical Engineering (SPIE)* (2002), vol. 4866, pp. 35–46.

[34] Mathis, M. Relentless Congestion Control. In *Proc. Int'l Wkshp on Protocols for Future, Large-scale & Diverse Network Transports (PFLDNeT'09)* (May 2009).

[35] Mathis, M., Semke, J., Mahdavi, J., and Ott, T. The macroscopic behavior of the TCP Congestion Avoidance algorithm. *Computer Communication Review 27*, 3 (July 1997).

[36] Menth, M., Schmid, M., Heiss, H., and Reim, T. MEDF - a simple scheduling algorithm for two real-time transport service classes with application in the UTRAN. In *Proc. IEEE Conference on Computer Communications (INFOCOM'03)* (Mar. 2003), vol. 2, pp. 1116–1122.

[37] Nichols, K., and Jacobson, V. Controlling Queue Delay. *ACM Queue 10*, 5 (May 2012).

[38] Pan, R., Piglione, P. N. C., Prabhu, M., Subramanian, V., Baker, F., and Ver Steeg, B. PIE: A Lightweight Control Scheme To Address the Bufferbloat Problem. In

High Performance Switching and Routing (HPSR'13) (2013), IEEE.

[39] RAMAKRISHNAN, K. K., FLOYD, S., AND BLACK, D. The Addition of Explicit Congestion Notification (ECN) to IP. Request for Comments 3168, RFC Editor, Sept. 2001.

[40] SALIM, J. H., AND AHMED, U. Performance Evaluation of Explicit Congestion Notification (ECN) in IP Networks. Request for Comments 2884, RFC Editor, July 2000.

[41] WELZL, M., AND FAIRHURST, G. The Benefits of using Explicit Congestion Notification (ECN). Internet Draft draft-ietf-aqm-ecn-benefits-08, Internet Engineering Task Force, Nov. 2015. (Work in Progress).

[42] WU, H., JU, J., LU, G., GUO, C., XIONG, Y., AND ZHANG, Y. Tuning ECN for Data Center Networks. In *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies* (New York, NY, USA, 2012), CoNEXT '12, ACM, pp. 25–36.

[43] XUE, L., CHIU, C.-H., KUMAR, S., KONDIKOPPA, P., AND PARK, S.-J. FaLL: A fair and low latency queuing scheme for data center networks. In *Intl. Conf. on Computing, Networking and Communications (ICNC 2015)* (Feb. 2015), pp. 771–777.

[44] ZHANG, T., WANG, J., HUANG, J., HUANG, Y., CHEN, J., AND PAN, Y. Adaptive-Acceleration Data Center TCP. *IEEE Transactions on Computers 64*, 6 (June 2015), 1522–1533.