Toke Høiland-Jørgensen

Bufferbloat and Beyond

Removing Performance Barriers in Real-World Networks

The elephant symbolises a big download, commonly referred to as an "elephant flow", which blocks the link with its bulk so everything has to wait behind it.

The mice are smaller flows, such as web pages, that take up little space but should be able to move freely. Through better queue management, the mice are guided around the elephant, so they don't have to wait; and interactivity is restored.

The penguin is Tux, the Linux mascot, who stands at the bottleneck and directs traffic onto the right path.

Bufferbloat and Beyond

Removing Performance Barriers in Real-World Networks

Toke Høiland-Jørgensen

Faculty of Health, Science and Technology

Computer Science

DOCTORAL THESIS | Karlstad University Studies | 2018:42

Bufferbloat and Beyond

Removing Performance Barriers in Real-World Networks

Toke Høiland-Jørgensen

DOCTORAL THESIS | Karlstad University Studies | 2018:42

Bufferbloat and Beyond - Removing Performance Barriers in Real-World Networks

Toke Høiland-Jørgensen

DOCTORAL THESIS

Karlstad University Studies | 2018:42

urn:nbn:se:kau:diva-69416

ISSN 1403-8099

ISBN 978-91-7063-878-7 (Print)

ISBN 978-91-7063-973-9 (pdf)

Revised version in which printing errors have been corrected

© The author

Distribution:
Karlstad University
Faculty of Health, Science and Technology
Department of Mathematics and Computer Science
SE-651 88 Karlstad, Sweden
+46 54 700 10 00

Print: Universitetstryckeriet, Karlstad 2018

WWW.KAU.SE

Bufferbloat and Beyond: Removing Performance Barriers in Real-World Networks

Toke Høiland-Jørgensen
Department of Mathematics and Computer Science
Karlstad University

Abstract

The topic of this thesis is the performance of computer networks. While network performance has generally improved with time, over the last several years we have seen examples of *performance barriers* limiting network performance. In this work we explore such performance barriers and look for solutions.

The problem of excess persistent queueing latency, known as *bufferbloat*, serves as our starting point; we examine its prevalence in the public internet, and evaluate solutions for better queue management, and explore how to improve on existing solutions to make them easier to deploy.

Since an increasing number of clients access the internet through WiFi networks, examining WiFi performance is a natural next step. Here we also look at bufferbloat, as well as the so-called *performance anomaly*, where stations with poor signal strengths can severely impact the performance of the whole network. We present solutions for both of these issues, and additionally design a mechanism for assigning policies for distributing airtime between devices on a WiFi network. We also analyse the "TCP Small Queues" latency minimisation technique implemented in the Linux TCP stack and optimise its performance over WiFi networks.

Finally, we explore how high-speed network processing can be enabled in software, by looking at the eXpress Data Path framework that has been gradually implemented in the Linux kernel as a way to enable high-performance programmable packet processing directly in the operating system's networking stack.

A special focus of this work has been to ensure that the results are carried forward to the implementation stage, which is achieved by releasing implementations as open source software. This includes parts that have been accepted into the Linux kernel, as well as a separate open source measurement tool, called Flent, which is used to perform most of the experiments presented in this thesis, and also used widely in the bufferbloat community.

Keywords: Bufferbloat, AQM, WiFi, XDP, TSQ, Flent, network measurement, performance evaluation, fairness, queueing, programmable packet processing

Acknowledgements

The journey is almost over. It has been challenging, exciting, annoying, incredible and above all *interesting*. And it would not have been the same without the incredible people who have accompanied and supported me along the way, all of whom deserve a huge thank you.

First of all, thank you to the many collaborators I have had these past years. Foremost among them of course my advisors, Anna Brunström and Per Hurtig, who have always been ready to offer competent feedback, ideas, suggestions and support; thank you so much to both of you! A special thanks also to Dave Täht, with whom I have collaborated extensively, on both the papers we have co-authored, and a list of other projects too long to reproduce here. Dave was also among the people who initially introduced me to the concept of bufferbloat, and he and Jim Gettys both played major roles in setting me on the right path, towards the work represented in this thesis; for this I am exceedingly grateful. Thank you, as well, to Jesper Brouer for his collaboration, and for convincing me to continue the work with Linux and open source post-thesis; and to Fanny Reinholtz for designing the awesome dust jacket for the printed version of this thesis. And finally, thank you to my other co-authors, and all the members of the Bufferbloat, make-wifi-fast, Linux and OpenWrt communities that I have collaborated with over the last six years.

A journey such as this would not have been possible without the friends who have shared my joys and sorrows along the way. Thank you to my colleagues at the university, my old friends in Denmark, and my new(er) friends in Karlstad. Thank you to Lea, Stefan, Stina and Leonardo for climbing, skiing and everything else. To Martin, Daniel, Alex and Miriam for hanging out and having fun. To Timo, Leo and Ricardo for bringing the band (back) together. To the innebandy team (in its various incarnations) for showing me how much fun it can be to chase a ball around with a stick. And to the dwarves, and to Sunni, Morten, Sidsel and Torben for keeping in touch through my five-year Swedish exile.

Last, but certainly not least, thank you so much to my wonderful loving family. Thank you mum and dad for helping me get to this point, for always being there for me, and for your love and respect. And thank you Sascha, for coming with me to Sweden and brightening up my days.

Karlstad, October 2018 — Toke Høiland-Jørgensen

Well then. A year and a half later, I find myself preparing a second print run; who would have thought? Many thanks to Dave Täht for insisting on this, and to Karlstad University for agreeing to print another batch, with all that entails.

Roskilde, April 2020 — Toke Høiland-Jørgensen

List of appended papers

I. Toke Høiland-Jørgensen, Bengt Ahlgren, Per Hurtig and Anna Brunstrom. Measuring Latency Variation in the Internet. ACM CoNEXT '16, December 12–15, 2016, Irvine, CA, USA.

II. Toke Høiland-Jørgensen, Per Hurtig and Anna Brunstrom. The Good, the Bad and the WiFi: Modern AQMs in a Residential Setting. Computer Networks, vol 89, pg 90–106, October 2015.

III. Toke Høiland-Jørgensen. Analysing the Latency of Sparse Flows in the FQ-CoDel Queue Management Algorithm. IEEE Communication Letters, October 2018.

IV. Toke Høiland-Jørgensen, Dave Täht and Jonathan Morton. Piece of CAKE: A Comprehensive Queue Management Solution for Home Gateways. IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN 2018), 25–27 June 2018, Washington, DC.

V. Toke Høiland-Jørgensen, Michal Kazior, Dave Täht, Per Hurtig and Anna Brunstrom. Ending the Anomaly: Achieving Low Latency and Airtime Fairness in WiFi. 2017 USENIX Annual Technical Conference (USENIX ATC 17), July 12–14 2017, Santa Clara, CA.

VI. Toke Høiland-Jørgensen, Per Hurtig and Anna Brunstrom. PoliFi: Airtime Policy Enforcement for WiFi. Under Submission.

VII. Carlo Augusto Grazia, Natale Patriciello, Toke Høiland-Jørgensen, Martin Klapez and Maurizio Casoni. Adapting TCP Small Queues for IEEE 802.11 Networks. The 29th Annual IEEE International Symposium on Personal, Indoor and Mobile Radio Communications (IEEE PIMRC 2018), 9-12 September 2018, Bologna, Italy.

VIII. Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern and David Miller. The eXpress Data Path: Fast Programmable Packet Processing in the Operating System Kernel. CoNEXT '18: International Conference on emerging Networking EXperiments and Technologies, December 4–7, 2018, Heraklion, Greece.

IX. Toke Høiland-Jørgensen, Carlo Augusto Grazia, Per Hurtig and Anna Brunstrom. Flent: The FLExible Network Tester. 11th EAI International Conference on Performance Evaluation Methodologies and Tools (VALUETOOLS 2017), December 5–7, 2017, Venice, Italy.

Comments on my participation

For all papers, the ideas have been developed in cooperation with my co-authors. However, I have done the bulk of all writing, implementation, experimental work and evaluation, with the following exceptions:

- In Paper I, the data analysis and most of the text describing the access network dataset was done by Bengt Ahlgren.
- In Paper IV, most of the initial implementation was done by Jonathan Morton. I have been responsible for the upstream submission to the Linux kernel, and have made major revisions in the course of this process. The design of CAKE has been heavily community-driven, with many members of the bufferbloat community taking part by testing and requesting features.
- In Paper V, a large part of the MAC layer queue rework implementation was done by Michal Kazior, and the 30 station tests were performed by Sven Eckelmann.
- In Paper VII, I was responsible for the ath9k and ath10k testing, and for submitting the final patch for inclusion in the upstream Linux kernel. The rest of the experiments and most of the writing was done by my co-authors.
- In Paper VIII, the design and implementation of XDP has been an ongoing process in the Linux development community, with most of the work being done by my co-authors. I have been responsible for designing and performing the experiments in the paper, together with Jesper Danggaard Brouer. I have also been responsible for writing and structuring the paper.
- In Paper IX the over-the-internet test run was done by Carlo Augusto Grazia.

Other relevant publications

The following is a list of the other relevant publications I have co-authored over the course of my PhD work, and how they relate to the work presented in this thesis:

- Jesper Dangaard Brouer and Toke Høiland-Jørgensen. “XDP - challenges and future work”. Linux Plumbers Conference 2018 Networking Track, Vancouver, Canada, November 2018. Here we present parts of the work described in Paper VIII.
- Toke Høiland-Jørgensen et al. “The Flow Queue CoDel Packet Scheduler and Active Queue Management Algorithm”. RFC 8290 (Experimental), January 2018. This is the standards document describing the FQ-CoDel algorithm also treated in Paper III.
- Toke Høiland-Jørgensen. “On The Bleeding Edge: Debloating Internet Access Networks”. Licentiate Thesis, Karlstad University, November 2016. This contains parts of the work presented in this thesis; specifically, Papers I, II and earlier versions of Papers V and IX as well as parts of the introductory summary.

- Toke Høiland-Jørgensen. "Bufferbloat mitigation in the WiFi stack - status and next steps". NetDev 2.2, Seoul, South Korea, November 2016. This includes early versions of the work presented in Papers V and VI.
- Toke Høiland-Jørgensen. "Flent: The FLExible Network Tester". The 11th Swedish National Computer Networking Workshop (SNCNW), Karlstad, Sweden, May 28–29, 2015. This is an earlier version of Paper IX.

Contents

Introductory Summary 1

1 Introduction 3

2 Research objective 4

3 Background 5
3.1 Bufferbloat and its mitigation 5
3.2 The 802.11 WiFi standards 6
3.3 High-performance software packet processing 9

4 Research questions 11

5 Contributions 13

6 Research method 16

7 Summary of appended papers 18

8 Conclusions and future work 22

Paper I:
Measuring Latency Variation in the Internet 31

1 Introduction 31

2 Datasets and methodology 32
2.1 The M-Lab NDT data 33
2.2 The access aggregation link data 34
2.3 Sources of latency variation 36

3 Latency variation over time and geography 36
3.1 Geographic differences 36
3.2 Development over time 38
3.3 Different measures of latency span 38

4 Latency variation in the access network 38

5 Examining queueing latency 40
5.1 Latency reductions after a drop 40
5.2 Delay correlated with load 42
5.3 Discussion 44

6 Related work 44
6.1 Large-scale active measurements 44
6.2 Targeted active measurements 45
6.3 Passive measurements 45

7 Conclusions 46

8 Acknowledgements 46

Paper II:
The Good, the Bad and the WiFi: Modern AQMs in a Residential Setting 51

1 Introduction 51

2 Related work 53

3 Experimental methodology 55

4 Tested algorithms 56
4.1 pfifo_fast 58
4.2 ARED 58
4.3 PIE 58
4.4 CoDel 58
4.5 SFQ 59
4.6 fq_codel 59
4.7 fq_nocodel 59

5 The Good: steady-state behaviour 60
5.1 The Real-time Response Under Load test 60
5.2 VoIP test 63
5.3 Web test 65
5.4 Discussion 69

6 The Bad: fairness and transient behaviour 69
6.1 Inter-flow fairness 69
6.2 Transient behaviour 72
6.3 Discussion 73

7 The WiFi: adding a wireless link 73
7.1 The RRUL test 75
7.2 VoIP traffic 76
7.3 Web results 77
7.4 Discussion 77

8 Conclusions and future work 77

Paper III: Analysing the Latency of Sparse Flows in the FQ-CoDel Queue Management Algorithm 87

1 Introduction 87

2 Related work 88

3 The sparse flow optimisation 89

4 Analytical framework 89
4.1 One sparse flow 89
4.2 Multiple sparse flows 91
4.3 Impact on bulk flows 92
4.4 Impact of changing the quantum 92

5 Real-world examples 92

6 Conclusion 95

7 Acknowledgements 95

Paper IV: Piece of CAKE: A Comprehensive Queue Management Solution for Home Gateways 99

1 Introduction 99

2 Background and Related Work 100
2.1 Bandwidth Shaping 101
2.2 Queue Management 101
2.3 DiffServ Handling 102
2.4 TCP ACK Filtering 102

3 The Design of CAKE 103
3.1 Bandwidth Shaping 103
3.2 Flow Isolation and Hashing 104
3.3 DiffServ handling 105
3.4 ACK filtering 106

4 Performance Evaluation 107
4.1 Host Isolation 107
4.2 DiffServ Handling 109
4.3 ACK Filtering 109

5 Conclusions 110

Acknowledgements 111

Paper V:
Ending the Anomaly: Achieving Low Latency and Airtime Fairness in WiFi 117

1 Introduction 117

2 Background 118
2.1 Bufferbloat in the context of WiFi 118
2.2 Airtime fairness 121

3 Our solution 123
3.1 A bloat-free queueing structure for 802.11 123
3.2 Airtime fairness scheduling 125
3.3 Implementation 128

4 Evaluation 129
4.1 Validation of effects 129
4.2 Effects on real-world application performance 135
4.3 Summary 138

5 Related work 139

6 Conclusion 140

7 Acknowledgements 140

Paper VI:
PoliFi: Airtime Policy Enforcement for WiFi 147

1 Introduction 147

2 Related work 148

3 The PoliFi Design 149
3.1 Userspace Policy Daemon 150
3.2 Weighted Airtime DRR 152
3.3 Kernel Airtime Scheduler 154

4 Evaluation 154
4.1 Steady state measurements 155
4.2 Dynamic measurements 157
4.3 DASH Traffic Test 159

5 Conclusion 159

PAPER VII:
Adapting TCP Small Queues for IEEE 802.11 Networks 163

1 Introduction 163

2 TCP Small Queues in a Nutshell 164

3 Controlled TSQ 167

4 Testbed 167

5 Results 171

6 Conclusions 176

PAPER VIII:
The eXpress Data Path: Fast Programmable Packet Processing in the Operating System Kernel 181

1 Introduction 181

2 Related work 183

3 The design of XDP 185
3.1 The XDP Driver Hook 186
3.2 The eBPF Virtual Machine 189
3.3 BPF Maps 190
3.4 The eBPF Verifier 191
3.5 Example XDP program 192
3.6 Summary 194

4 Performance evaluation 194
4.1 Packet Drop Performance 196
4.2 CPU Usage 197
4.3 Packet Forwarding Performance 198
4.4 Discussion 199

5 Real-world use cases 200
5.1 Software Routing 201
5.2 Inline DoS Mitigation 202
5.3 Load Balancing 203

6 Future directions of XDP 204
6.1 Limitations on eBPF programs 204
6.2 User Experience and Debugging 205
6.3 Driver Support 205
6.4 Performance Improvements 205

6.5 QoS and Rate Transitions 206
6.6 Accelerating Transport Protocols 206
6.7 Zero-copy to userspace 206
6.8 XDP as a building block 207

7 Conclusion **207**

Paper IX:
Flent: The FLExible Network Tester **215**

1 Introduction **215**

2 Experimental challenges **216**
2.1 Reproducing experiments 216
2.2 Testbed configuration and test automation 217
2.3 Storing and analysing measurement data 217

3 How Flent helps **217**
3.1 Reproducing experiments 218
3.2 Configuration and automation 218
3.3 Storing and analysing measurement data 219

4 Showcasing Flent: A look at the BBR congestion control **219**
4.1 Experimental setup 220
4.2 Testbed results 221
4.3 Public internet results 224

5 Related work **227**

6 Conclusions and Future Work **227**

Introductory Summary

"To Infinity… And Beyond!"

Buzz Lightyear, Toy Story

1 Introduction

Internet usage is steadily increasing and our world is becoming increasingly connected. At the same time, new applications place ever higher performance requirements on the network, which must support both high bandwidth and low latency. Latency in particular has become more important as applications turn more interactive, and emerging applications such as augmented reality and ubiquitous video conversations are only going to make this more important.

While network performance has increased significantly, and continues to do so, over the last several years we have seen examples of *performance barriers* limiting the network performance. One example of this is the *bufferbloat* phenomenon, which is a term coined to describe the effect that occurs when a network bottleneck is congested and large buffers fill up *and do not drain*, thus inducing a persistent queueing delay that can be much larger than the path round-trip time [1, 2]. Excessive queueing delay is by no means a new phenomenon, but even so, bufferbloat has been found to be widespread in deployed systems, thus forming a practical performance barrier.

In this thesis we will explore such performance barriers, and look for ways to remove or mitigate them. Bufferbloat serves as our starting point. An online community has formed around the need to develop technical solutions to mitigate bufferbloat,[1] and the work in this thesis should be seen in the context of this community effort, which I have taken part in over the last several years.

The bufferbloat effort has been focused on last-mile access networks, where bufferbloat has been both extremely prevalent, but also fixable due to cheap home routers that can have their firmware replaced. Replacing the router firmware makes it possible to apply state of the art queue management algorithms on the bottleneck link connecting the home to the internet. The efficacy of such queue management solutions is one of the topics we examine in this work.

As we improve the access link itself, and as bandwidths increase, the new bottleneck link, and source of performance barriers, becomes the WiFi link. Thus, exploring the performance of WiFi networks is a natural extension of the bufferbloat work. It turns out that WiFi network performance suffers not only from bufferbloat, but also from the so-called *performance anomaly*, where stations with poor signal strengths can severely impact the performance of the whole network. We explore a range of performance issues with WiFi networks and present solutions for them.

Both the solutions for bufferbloat and the improvements to WiFi networks are characterised by being enabled by malleable open source software implementations which can be inspected and improved upon. This has been possible in the relatively low-speed home network devices where CPUs can keep up with the packet rate. However, for truly high-speed networking (going into tens and hundreds of gigabits per second), programmable packet processing has been out of reach until quite recently. Now, not only is special-purpose

[1] See `https://www.bufferbloat.net`.

networking hardware gaining programmability, general-purpose computers are also beginning to reach a level of performance where they can realistically process packets at the 100-Gbps level. The software architecture of common operating systems forms a bottleneck when trying to achieve such speeds, which is the last barrier that we explore solutions to in this work. Specifically, we look at the eXpress Data Path framework that has been gradually implemented in the Linux kernel over the last several releases, which we describe and evaluate in detail.

This thesis explores the performance barriers mentioned above, in the form of the nine appended papers. Before we get this far, though, we will first set the stage and present some context. This is the role of this introductory summary, the rest of which is structured as follows: Section 2 (below) outlines the main research objective we are pursuing in this thesis. Section 3 contains a short background primer on the subject areas we touch upon, and summarises related work. Section 4 outlines our research questions and Section 5 outlines the main contributions of this thesis. Section 6 relates the work to the traditions of the field of computer science and Section 7 presents a short summary of the appended papers. Finally, Section 8 concludes and outlines future work.

2 Research objective

The overarching research objective of this work is to improve network performance by exploring specific performance barriers in modern real-world networks, and developing ways of removing them. We use the bufferbloat phenomenon as a starting point for this exploration and continue into several related areas. In particular, we are interested in the following:

1. Improving the understanding of the specific manifestations of bufferbloat and of its prevalence in the internet.
2. Evaluating the efficacy of existing solutions for bufferbloat, and identifying ways to optimise them for deployment in edge networks.
3. Addressing bufferbloat in WiFi networks and exploring other related performance problems and their solutions.
4. Exploring solutions for removing the performance barrier posed by the operating system in software packet processing systems.

3 Background

This section provides some background on the subject areas of this thesis. Section 3.1 outlines the previous work on bufferbloat assessment and mitigation; Section 3.2 gives an overview of the operation of 802.11 WiFi networks, and outlines previous work on network performance in the WiFi space; and Section 3.3 outlines previous work on high-performance packet processing in software.

3.1 Bufferbloat and its mitigation

While bufferbloat is straightforward to demonstrate in a lab setting, there has been some argument over how widespread it is in the real world. Due to its distributed nature, no one has a global view of the internet, so one is limited to analysing data from a subset of the network. This can be done either by generating measurement traffic to look for specific behaviour (active measurements), or by capturing traffic at a specific point in the network and trying to infer path characteristics from that (passive measurements).

Several studies based on both active and passive measurements have been performed, with various focus areas. Examples of large scale studies that include latency measurements are [3], which uses 130,000 tests from the Netalyzr test suite to measure queueing latency; [4], which uses data from the BISMark and SamKnows measurement platforms to measure baseline and under-load latency; and [5], which also uses BISMark data (as well as other sources) to measure broadband performance in South Africa. Smaller scale active measurements are described in [6], where active probing of residential hosts is used to measure network connections; in [7], which is specifically targeted at assessing queueing latency; and in [8, 9], both of which use clients under the experimenters' control to measure bufferbloat in cellular networks. Fewer passive measurement studies specifically look at latency, but a few examples can be found, e.g. [10, 11], both of which employ captures taken at a network edge for their analysis.

Several of the studies mentioned above find examples of latency increases over the baseline, but few of them specifically discuss bufferbloat as a possible cause. In Paper I we seek to contribute to the discussion, by exploring specifically how latency varies over time in the internet, and how much of this can be attributed to bufferbloat, with the data available.

Turning to bufferbloat mitigation procedures, the problem of unmanaged FIFO queues and their impact on latency has been known for a long time, and the introduction of Active Queue Management (AQM) has been known to be an effective way of reducing delays, if deployed correctly. The most well-known classical AQM algorithm, Random Early Detection (RED) [12], is more than two decades old, and its successor, Adaptive RED (ARED) [13], is over fifteen. Package scheduling algorithms, which can also help mitigate queueing latency, have even older well-known examples, such as the Stochastic

Fairness Queueing (SFQ) algorithm [14], and other fairness queueing algorithms that schedule individual flows to ensure fairness between them.

Many other algorithms have been proposed over the years [15], however few have seen widespread deployment in the internet. This is mostly due to the difficulty of tuning the algorithms, and the adverse effect they can have on performance if configured incorrectly [16]. And so, for many years no easily deployable queue management techniques were readily available.

As the bufferbloat issue started receiving increased attention, however, we started seeing a resurgence of interest in developing algorithms that can effectively control queues without requiring extensive tuning. And so, a number of new algorithms appeared: Controlled Delay (CoDel) [17] and Proportional Integral controller Enhanced (PIE) [18] are both new AQM algorithms, while FQ-CoDel [19] combines the CoDel AQM with a fairness queueing scheme and an optimisation for improving the latency of "sparse" flows (i.e., flows that don't use a lot of bandwidth). These algorithms are the focus of the evaluation presented in Paper II.

To supplement the experimental evaluation of different queueing algorithms, an analytical approach can be useful to explore specific performance characteristics of an algorithm. There have been examples of this, such as that performed for the Shortest Queue First (SQF) algorithm by its authors in [20], or for the Quick Fair Queueing (QFQ) algorithm in [21]. Another example of such an analytical approach is the analysis of the number of active flows in a fairness queueing system that is provided in [22]. However, no such analysis has been performed for the new queue management algorithms discussed above. Remedying this is the subject of Paper III.

While effective algorithms for controlling bufferbloat are certainly necessary to solve the problem, they are not themselves sufficient. A queue management algorithm is only effective if it is in control of the bottleneck queue. This can be achieved by eliminating queues in lower layers, such as has been achieved in Linux for Ethernet drivers [23]. If this is not possible, the bottleneck queue can be moved through the use of a bandwidth shaper. The most common way to do this is by a token bucket-based shaper (e.g., [24]), or by a rate-based shaper (e.g., [25]). Integrating bandwidth shaping with a queue management algorithm complicates deployment, which can be a barrier to adoption. Exploring how to improve on this situation is the topic of Paper IV.

3.2 The 802.11 WiFi standards

The IEEE 802.11 standards [26] are a series of documents that describe the physical and MAC layers of the wireless network protocol that is commonly known as WiFi. The protocol acts like a virtual Ethernet link, but because the radio waves are propagating through a shared medium, there are several important differences in how the protocol operates. In this section, we summarise the main operational principles of the 802.11 WiFi MAC. We omit the physical layer and focus on the 802.11n revision of the specification, which is what we have used in Paper V and Paper VI. The newer 802.11ac revision

mainly differs at the physical layer, and in some constants (such as the max aggregation size) at the MAC layer. The overall operational principles are almost identical between 802.11n and 802.11ac.

When a node in a WiFi network (or "station", as non-access point nodes are typically called) has data to send, it will first attempt to determine whether or not the channel is free, by listening for other transmissions. If it determines that no other devices are currently transmitting, it will wait a random amount of time (in the order of several microseconds), and then begin transmitting its data frame. After transmission has completed, it will wait for the receiving device to acknowledge the transmission. If no acknowledgement is heard, the sender will assume that a collision occurred (i.e., that another sender transmitted at the same time and the transmissions interfered with each other and were lost), and so will initiate another, longer, random wait period, before the procedure is repeated. This mechanism of arbitration for the opportunity to send is known as the 802.11 Distributed Coordination Function (DCF). The standard additionally specifies several possible transmission rates (with corresponding physical encodings of the data), and leaves it up to the devices to pick the best rate for a given transmission opportunity, based on the observed signal-to-noise ratio at the time of transmission.

The main advantage of the DCF is that it does not require a central coordinator in the network. However, this comes at a cost: The random back-off periods can lead to inefficiency in the network (i.e., periods of time in which the channel is entirely idle). To limit the overhead of the DCF, aggregation has been introduced into the MAC protocol, allowing several data packets to be sent at once when a device gets a chance to transmit. The standard specifies a maximum size and a maximum transmission time for aggregates, but not how devices should build aggregates. From a queueing latency perspective, however, it is clear that some amount of queueing is necessary in order to have enough packets ready to build a reasonably sized aggregate when given a chance to transmit. The introduction of aggregation also makes the transmission bursty (there will be periods of no data being transmitted, followed by short bursts of several packets when an aggregate is formed). This introduces its own challenges for a low-latency queueing structure, as we will see in Paper V.

Another fundamental property of the DCF is that it gives each device on the network the same probability of winning the contention for the medium, and thus being able to transmit. However, this is independent of the time each device spends transmitting when it gets a chance. Since the transmission rate can vary over a large interval in the same network, one station transmitting at a low rate can take up a large fraction of the total transmission time, and so effectively limit the throughput of all stations in the network, which hurts the aggregate efficiency of the network as a whole. This phenomenon is known as the 802.11 performance anomaly [27].

The final property of WiFi networks worth mentioning here is the Quality of Service (QoS) standards defined by the 802.11e document in the 802.11

standards series. This specifies that a station can transmit data in different priority tiers depending on their QoS markings. There are four QoS levels, labelled (in decreasing priority order) as Voice, Video, Best Effort and Background. The standard specifies different constants for the DCF for each QoS level, giving the higher priority levels a higher probability of winning the contention for the transmission medium. The standard also specifies different aggregation size limits on the different QoS levels, which in theory prevents abuse by trading higher contention win probability for lower maximum throughput (due to the lower aggregation). From a latency perspective, the QoS mechanism can in theory be a big win, but unfortunately there are no agreed-upon standards for how to assign QoS levels to different traffic flows, and implementations tend to be bug-ridden, significantly lowering the usefulness of this feature of the standard.

While there is nothing fundamental in WiFi stating that queues cannot be managed to avoid excess queueing latency, there is little in the literature that deals with this aspect of the problem. A few examples of work that do treat queueing latency in WiFi networks are [28], which features a WiFi component in a larger evaluation of bufferbloat mitigation techniques; [29], which looks at buffer sizing issues in WiFi networks; and [30], which touches upon congestion in the WiFi hop and uses different queueing schemes to address it, but in the context of a centralised solution that also seeks to control fairness in the whole network. However, none of these studies provides a solution at the WiFi hop itself.

The performance anomaly, on the other hand, has been studied extensively, and several solutions have been proposed, falling roughly into three categories: Those that modify the back-off parameters of WiFi nodes to achieve fairness [31–34], those that change the transmission size [35–37], and those that employ a scheduler at the access point [38, 39]. However, most of the solutions have never made it to actual implementations, whether because the implementation has never left the simulation stage, or because the approach has other tradeoffs that make them impractical to implement. This means that the performance anomaly is still very much present today, as we show in Paper V.

The decentralised nature of WiFi also means that it lacks protocol support for enforcing policies on resource usage (notably airtime usage), which would otherwise be an obvious way to increase performance for some applications, especially in contested scenarios. There are, however, some attempts at applying policies to WiFi networks [40–42] but none of them are at a state where they can realistically be deployed on today's networks. Some enterprise access points do offer features related to airtime fairness and policy configuration [43], but being a proprietary system, not much information is available for study of the mechanisms involved. How to design a workable policy enforcement mechanism for WiFi is discussed in Paper VI.

3.3 High-performance software packet processing

Packet processing in the network has historically been limited to the most basic operations, simply because high traffic volume means very tight bounds on the time spent processing each packet. However, as hardware becomes more capable, and costs drop, it becomes increasingly feasible to perform advanced processing as part of the data path. This has lead to an increasing demand for programmable capabilities in networking hardware. One example of this is the advent of of programmable hardware devices, such as the NetFPGA [44], which enables custom packet processing on specialised hardware, by exposing an API that makes it possible to run arbitrary packet processing tasks on the FPGA-based dedicated hardware. Another example is the P4 language [45], which seeks to extend this programmability to a wider variety of packet processing hardware.

However, even though specialised hardware has gained data plane programmability, it still cannot match the flexibility and low cost of common off-the-shelf (COTS) hardware. Several packet processing systems running on COTS hardware have appeared over the last several years. Examples of this include applications performing single functions, such as switching [46], routing [47], named-based forwarding [48], classification [49], caching [50] or traffic generation [51]. They also include more general solutions which are highly customisable and can operate on packets from a variety of sources [52–57].

One challenge with using COTS hardware, is that in order to achieve high performance it is necessary to remove any bottlenecks between the networking interface card (NIC) and the program performing the packet processing. Since one of the main sources of performance bottlenecks is the interface between the operating system kernel and the userspace applications running on top of it (because of the high overhead of a system call and complexity of the underlying feature-rich generic networking stack in the operating system), low-level packet processing frameworks have to manage this overhead in one way or another. The approaches taken to this fall into three broad categories: (a) implementing parts of the application as a module inside the kernel itself, with examples such as the Open vSwitch [57] virtual switch and the Click [55] virtual router framework; (b) providing an interface for userspace to access packet data with lower overhead than with traditional sockets, such as in PF_RING [58], the Packet I/O engine that is part of PacketShader [47] or the Netmap [59] framework, as well as special-purpose operating systems such as Arrakis [60] and ClickOS [61]; or (c) bypassing the kernel entirely and handing over control of the networking device directly to userspace, where examples include the PF_RING ZC module [62], the hardware-specific Solarflare OpenOnload [63] or, most prominently, the DataPlane Development Kit (DPDK) [64].

Each of the three categories of solutions have different drawbacks and tradeoffs, and while all of them offer better performance than the regular operating system network stack, the relative performance of each solution varies widely. The highest raw performance numbers are seen with kernel bypass

solutions, of which DPDK represents the state of the art. DPDK is commonly used with higher-level processing frameworks such as Cisco's VPP [54], which offers a programming model that helps maximise performance for many applications.

Because kernel bypass solutions need direct access to the hardware to achieve the high performance numbers, it is difficult to integrate with systems where the kernel plays a dominant role in resource abstraction and isolation, such as container-based and virtualised workloads. A different solution, that allows more fine-grained integration into the operating system, has been developed by the Linux kernel community over the last several years. This solution is called XDP, but has been lacking a high-level architectural description. Remedying this lack is the topic of Paper VIII.

4 Research questions

In order to achieve the overall research objectives outlined in Section 2, we formulate the following research questions that we seek to answer in this work:

1. *How bloated is the internet?*

   Before one can address a problem, one must first be aware that it exists. In particular, before spending the considerable resources required to deploy upgrades to the edge of the internet (with its billions of connected devices), one must be convinced of the gain. While the bufferbloat problem can be easily demonstrated in a lab, there has been some debate over how prevalent it is in the real world (as discussed in the previous section). Since no one has a global view of the internet, finding the right answer is no trivial task. However, we can make a dent in the problem by analysing the data we **do** have available, and by looking at smaller scale representative slices of the network. In this work we set out to do just that, and thereby further the understanding of the problem.

2. *Can we solve bufferbloat by smarter queue management?*

   Knowing that there is a problem to be solved, one naturally starts looking for solutions. In the previous years, several possible solutions to the bufferbloat issue have been proposed, often in the form of queue management and packet scheduling algorithms that seek to ensure low latency even when a link is heavily loaded.

   However, such algorithms are often evaluated only by their inventors, in scenarios that reflect the design goals of the algorithms. Thus, comparative studies of several algorithms are needed to fully assess their relative efficacy. We set out to provide such a comparison, thereby answering the question of how well the bufferbloat issue can be resolved today. This experimental comparison clearly shows that the FQ-CoDel algorithm can offer tremendous performance benefits, especially for low-rate flows that do not build a large queue themselves. We seek to examine this in more detail by supplementing our experimental results with an analysis of the conditions under which flows can benefit from the low latency guarantees of the algorithm, and what exactly those guarantees are.

   Finally, we seek to improve the deployment aspects of new bufferbloat solutions. An important aspect of this is ease configuration, especially in devices at the edge of the internet, where the problem tends to be worse. Often, several different parts, such as queue management and bandwidth shaping, need to be integrated into a solution before it becomes deployable. To ease such deployment, we seek to develop a solution that integrates state of the art algorithms into an easy to deploy solution for bufferbloat at the network edge.

3. *How can we improve the performance of WiFi?*

   As explained in the previous section, two major performance issues affect modern WiFi networks: The bufferbloat issue, and the performance anomaly. We set out to find workable solutions for both these issues, using the Linux kernel as our example platform. In addition, we set out to explore how these solutions can be used to implement another performance-enhancing feature: Policy-based airtime assignment to devices on a WiFi network. Finally, we set out to improve the way WiFi networks interact with other bufferbloat mitigation solutions implemented in Linux, focusing on the TCP Small Queues optimisation in the TCP stack.

4. *How can we integrate high-performance programmable packet processing into the operating system?*

   As outlined in the previous section, while several solutions for high-performance packet processing exist, to avoid the performance penalty imposed by interacting with the operating system kernel for every packet, they all either heavily modify the operating system, or bypass it entirely. However, this limits flexibility in packet processing and makes it harder to integrate it into a mixed environment where applications relying on the regular operating system environment is also present. We seek to investigate how the performance barrier of the operating system can be lifted, allowing programmable packet processing to be integrated into the operating system in a cooperative way.

5 Contributions

The main contributions of this work are the following:

1. *A better understanding of the magnitude of bufferbloat in the internet*

   We contribute to a better understanding of bufferbloat and its magnitude in the internet. We do this by combining a large scale active measurement dataset from the public internet with a smaller scale passive measurement dataset from an internet service provider access network. We show that significant latency variation occurs, with large differences between regions, but negligible development over time, despite consistent increases in bandwidth over the same time span. In addition, we use the observed variation in latency as a way to quantify excess latency, and combine it with a novel approach to identify bufferbloat from TCP RTT samples, which we employ on a subset of the data to show that at least some of the latency variation can be attributed to bloat.

2. *An evaluation of existing solutions for bufferbloat mitigation*

   We contribute an extensive experimental evaluation of several modern queue management algorithms in a Linux testbed setup, designed to model a home network setup. This evaluation provides important real-world performance data on modern queue management algorithms. Our evaluation covers a range of simulated access network connection speeds, as well as a WiFi bottleneck scenario. We show that modern queue management algorithms can significantly reduce bufferbloat at the access link; however, the tested AQM algorithms have some issues with transient delay spikes as flows start up, and they exacerbate TCP flow unfairness, while the tested fairness queueing algorithms provide consistently low latency and almost perfect fairness. Our web and VoIP application tests show that the improvements in latency afforded by the algorithms translate to real-world application performance benefits. Finally, we show that the algorithms are less effective on a WiFi bottleneck, due to buffering in lower layers of the operating system network stack.

3. *An analysis of the sparse flow behaviour of FQ-CoDel*

   Our evaluation of queue management algorithms clearly shows that the FQ-CoDel algorithm achieves the lowest latency for many workloads, because of its optimisation for prioritisation of flows that do not build a queue, so-called "sparse flows". We contribute to the understanding of this mechanism by performing an analysis of which conditions a packet flow must fulfil to be considered "sparse" by the FQ-CoDel queue management algorithm, and thus given priority and low latency. We combine analytical expressions derived from the algorithm description with a numerical simulation, and formulate conditions that a flow needs to fulfil to keep the "sparse" status, as well as expressions for the expected latency of such a flow.

4. *An integrated queueing solution targeting home network gateways*

   We present an integrated queue management solution for home network gateways. This solution, called CAKE, builds on the basic fairness queueing design of FQ-CoDel, but adds several features are useful in a home gateway. These features include traffic shaping, host-based fairness queueing, DiffServ handling and TCP ACK filtering. Our evaluation shows that these features provide compelling benefits for their respective use cases. In addition, the integration of these features into a single queueing discipline significantly eases deployment and configuration, which is an important aspect of defeating bufferbloat in real-world networks.

5. *A solution for improving the performance of WiFi*

   We design and implement workable measures to resolve both the bufferbloat issue and the performance anomaly in WiFi, using Linux as an implementation platform. Our solution reduces latency in the stack by an order of magnitude by improving queue management, and it increases the efficiency of the network with up to a factor of five by enforcing airtime fairness between devices. We have worked with the Linux community to get our solution incorporated into the mainstream Linux kernel, to make sure it is not just a theoretical solution, but rather one that will find its way into deployed devices. Our solution builds on state of the art queue management algorithms, and showcases new ways of adapting these techniques to the WiFi domain.

6. *An airtime policy enforcement system for WiFi*

   Building on our airtime fairness system for solving the performance anomaly, we design and implement a solution for enforcing configurable airtime policy assignment between stations in an infrastructure WiFi network. This enables several interesting use cases, from a limited guest network use case, to a full network slicing solution as is being discussed for future 5G network environments. We implement the system in Linux and show how it can be used to improve performance of real-world applications.

7. *An improvement to the TCP Small Queues mechanism for WiFi networks*

   The "TCP Small Queues" feature included in the Linux kernel limits the size of the queue the TCP stack keeps for each flow, in order to reduce queueing latency. We show that this has an adverse impact on the performance of TCP over WiFi, because there is not enough data queued to build aggregates at the WiFi MAC level. We analyse the magnitude of this degradation and implement a solution which selectively tunes the TCP Small Queues system when flow egress is over a WiFi link.

8. *A description and evaluation of the eXpress Data Path programmable packet processing system in Linux*

   The Linux kernel networking community have designed and implemented the eXpress Data Path system in the Linux kernel over the last several years. This system allows high-performance packet processing to be integrated into the kernel in a cooperative way, increasing flexibility compared to other systems, while retaining high performance. We contribute a description of the architecture of the XDP system, along with a thorough performance evaluation featuring both synthetic benchmarks and real-world application examples. We make the case that XDP presents a compelling tradeoff in terms of features, flexibility and performance compared to other related systems.

9. *A tool for automation and re-usability of experiments*

   In the course of answering the main research questions and developing the contributions outlined above, we have developed and described a testing tool that helps facilitate future experimental work, by making tests repeatable and results sharable. The tool is called The FLExible Network Tester (Flent), and contains several new features for easy data exploration and automation of experimental work. Flent has shown real-world utility and is widely used in the bufferbloat community.

6 Research method

This thesis is written within the field of computer science, more specifically in the area of computer networking. Computer science, in turn, is a part of the broader field of *computing*. This field is quite diverse, but can be viewed as being comprised of three different aspects, each drawing on different traditions, namely mathematics, engineering and science [65]. In this section, we situate the work presented in this thesis in the broader field of computing and these three aspects that comprise the field.

The mathematical origins of computing are tied to the theoretical foundations of the field, most notably the theory of computation and the question of what can be computed. In addition, mathematical analysis and proof theory plays an important role in many branches of computing. The engineering aspect has also played a significant role throughout the history of computing, simply because the practical realisation of computation (i.e., the building of actual computers) has been an integral part of the development of the field. Today, the engineering influences is most clearly seen in the term "software engineering". Finally, "computer science" is commonly used when naming university departments (as is the case at Karlstad University), but determining what exactly it is a science *of* has been the subject of quite some discussion.

As mentioned above, the research presented in this thesis concerns computer networks and their function. It wasn't long after the first computers were built that it became evident that it was useful to have different computers communicate with each other. This soon led to the development of the Arpanet, which in turn has developed into the modern internet [66]. Today, the internet has become such an integral part of society that many functions cannot exist without it. Like the rest of the field of computing, networking rests on the three legs of mathematical theory (e.g., queueing theory, distributed algorithms, etc.), engineering (the internet as a whole is by far the largest and most complicated machine built by humankind) and science (where the network itself can become the subject of study).

The research presented in this thesis contains elements of all three aspects comprising the broader field. Mathematical analysis of an algorithm is the primary focus of Paper III, and is also incorporated into Paper V, where it serves to inform the design and implementation of the presented solution.

The engineering aspect is present in most of the work presented here. Indeed, an important motivation of the work has been to ensure the ideas are developed all the way to the point where they can be implemented in real systems, and thus improve today's networks. The primary avenue for this has been open source software, most notably the Linux kernel. The solutions developed in Papers IV, V, VI, VII and VIII are all included in the upstream Linux kernel, or are in the process of being incorporated at the time of writing. And the Flent tool presented in Paper IX is released as its own open source project, and has been used in the bufferbloat community for years.

Finally, the aspect of scientific study is included where the functioning of the network itself becomes the object of study. In networking, this is

usually achieved in one of three ways: By simulation (where a computer program will simulate an entire network and the packets flowing through it), emulation (where the network is simulated, but interacts with, e.g., real operating system network stacks) and experiments (where real networking equipment and software is used in the experiments). All too often, new concepts in networking are never developed past the simulation stage, which means they languish unused and do not make it to the stage where they can be deployed on the real internet. In addition, simulation is built on idealisations which get ever further from reality as optimisations are added to networking devices, often leading to very different behaviour than those assumed by the simulation tools. To counteract this, the focus in this thesis is very deliberately on the latter two kinds of study. Papers I and II are examples of this, while many of the others include experimental studies as part of the evaluation.

7 Summary of appended papers

Paper I – Measuring Latency Variation in the Internet

In this paper we examine two complementary datasets to assess to what extent bufferbloat is a real problem in the internet. We do this by analysing latency variation in a large-scale active measurement dataset from the Measurement Lab Network Diagnostic Tool, combined with a passive measurement data set from an access link.

The former dataset allows us to look at large scale trends, but because it consists of data from active measurements performed over the public internet, we can only use it to infer the potential for bufferbloat, not the frequency with which it occurs. The other dataset is much smaller in scale, but is based on passive measurements and comes from a network that has known path characteristics. This means that we can make conclusions about what the data shows with higher certainty. The combination of these two datasets allows us to say something meaningful of the latency characteristics of the internet as a whole.

We find that significant latency variation is present in both datasets. Additionally, a more detailed analysis of a subset of the data shows that at least some of it can be attributed to bufferbloat.

Paper II – The Good, the Bad and the WiFi

In this paper we evaluate a selection of bottleneck queue management schemes in a test-bed representative of residential internet connections of both symmetrical and asymmetrical bandwidths as well as WiFi. Latency under load and the performance of VoIP and web traffic patterns are evaluated under steady state conditions. Furthermore, the impact of the algorithms on fairness between TCP flows with different RTTs, and also the transient behaviour of the algorithms at flow startup is examined.

We show that the tested AQM algorithms can significantly improve the steady state performance, but that they exacerbate TCP flow unfairness and severely struggle to contain queueing delay in transient conditions, such as when flows start up. The tested fairness queueing algorithms, on the other hand, almost completely eliminate these problems and give consistently low latency and high throughput in the tested scenarios.

Finally, we show that all the tested algorithms perform worse on a WiFi bottleneck because they are limited by significant queueing in lower layers of the stack, and thus outside the control of the algorithms.

Paper III – Analysing the Latency of Sparse Flows in the FQ-CoDel Queue Management Algorithm

In this paper we analyse an aspect of the FQ-CoDel queue management algorithm that has thus far not been well-explored: The conditions under which a particular flow is considered "sparse", and thus receives preferential treatment by FQ-CoDel. We formulate a set of constraints that a sparse flow must satisfy, and also formulate a set of expressions for the expected queueing latency of sparse flows.

To verify the analytical expressions, we use a numerical example to show how many of a given type of sparse flows (Voice over IP traffic) a given link can support, and verify the analytical predictions in a numerical simulation. We show that the number of sparse flows that a given bottleneck can service with low latency is only dependent on the number of backlogged bulk flows at the bottleneck. Furthermore, we show that as long as the maximum number of sparse flows is not exceeded, all sparse flows can expect a very low queueing latency through the bottleneck.

Paper IV – Piece of CAKE

In this paper we present the design and implementation of the CAKE queue management algorithm, which is a *comprehensive network queue management system* designed specifically for home internet gateways. CAKE is built upon the FQ-CoDel hybrid AQM and flow queueing algorithm, but adds several features targeted specifically at home gateways while being easy to configure and deploy. These features include: bandwidth shaping with overhead compensation for various link layers; reasonable DiffServ handling; improved flow hashing with both per-flow and per-host queueing fairness; and filtering of TCP ACKs.

Our evaluation shows that each of these features offer compelling advantages for their respective use cases, giving CAKE the potential to significantly improve the performance of last-mile internet connections. CAKE has been accepted into the upstream Linux kernel and is included from Linux v4.19, released in October 2018.

Paper V – Ending the Anomaly

In this paper we present workable solutions to both bufferbloat at the WiFi link and the 802.11 performance anomaly. We implement a queueing scheme in Linux that is based on FQ-CoDel and tightly integrated with the MAC layer to solve the bufferbloat issue, and a scheduler-based solution to achieving airtime fairness.

We formulate an analytical model for achievable 802.11n throughput with and without airtime fairness and use that to evaluate our solution in combination with a series of testbed experiments. We achieve an order of magnitude reduction in latency under load, large improvements in multi-station throughput,

and nearly perfect airtime fairness for both UDP and TCP traffic. Further experiments with application traffic confirm that our modifications provide a significant performance gain for real-world traffic.

Paper VI – PoliFi

In this paper we present the design and implementation of PoliFi, an airtime policy solution for WiFi. PoliFi builds on the previous work presented in Paper V, but generalises the airtime fairness solution and adds a mechanism for specifying policies for how airtime is assigned to different stations. In addition, it includes a policy daemon that enforces the user-configured policies in real time, and is able to implement a variety of useful policies. These include prioritisation of specific devices; balancing groups of devices for sharing between different logical networks or network slices; and limiting groups of devices to implement guest networks or other low-priority services.

Our evaluation shows that that PoliFi successfully enforces the desired policies; and we show how these can be used to improve the performance of a real-world DASH video streaming application.

Paper VII – Adapting TCP Small Queues for IEEE 802.11 Networks

In this paper we examine the interactions between WiFi networks and the TCP Small Queues (TSQ) mechanism that is a part of the Linux TCP stack and is designed to keep queues small for TCP sockets originating on the local machine. We show that the default settings for TSQ prevent enough data from being buffered to properly utilise the packet aggregation mechanism in WiFi links. We evaluate a range of tuning parameters for 802.11n and 802.11ac and show that it is possible to double the throughput for 802.11n and increase it by an order of magnitude for 802.11ac, with a negligible increase in queueing latency.

Paper VIII – The eXpress Data Path

In this paper we describe the eXpress Data Path (XDP) programmable packet processing framework that has been introduced into the Linux networking stack over that last several years. XDP represents a novel approach to programmable packet processing, where the processing is integrated into the kernel device drivers in a cooperative way, by way of an execution environment that executes custom byte code which the kernel statically analyses for safety, and translates into native instructions.

This approach allows user programs to selectively process some packets, while letting the operating system networking stack handle others. In addition, packet processing programs can make use of kernel features to perform certain operations, and access state from both the networking stack, userspace programs and other parts of the kernel. Together, these features offer compelling

advantages by enabling programmable packet processing without the all-or-nothing approach imposed by user-space networking applications that need to take over the networking device entirely.

Our evaluation shows that XDP achieves single-core packet processing performance as high as 24 million packets per second. We also illustrate the flexibility of the programming model through three example use cases: layer-3 routing, inline DDoS protection and layer-4 load balancing.

Paper IX – Flent

In this paper we present a tool designed to make experimental evaluations of networks more reliable and easier to perform. This tool, called Flent, works by composing well-known benchmarking tools to, for example, run tests consisting of several bulk data flows combined with simultaneous latency measurements. Tests are specified in source code, and several common tests are included with the tool. In addition, Flent contains features to automate test runs, and to interactively plot and explore data collected from experiments.

8 Conclusions and future work

In this thesis we have explored a range of performance barriers that can limit real-world network performance, and identified solutions for them. The first such issue is bufferbloat, where we have contributed to understanding the extent of the problem through an assessment of the magnitude of excess latency in the public internet; performed a thorough evaluation of existing queue management solutions targeted at mitigating bufferbloat; and analysed the performance characteristics of the state of the FQ-CoDel algorithm in detail. Following this, we have shown how deployment of bufferbloat solutions can be made easier through the CAKE integrated queue management and traffic shaping system.

Having identified WiFi networks as an area where existing bufferbloat solutions fell short, we have designed and implemented a solution for bufferbloat tailored specifically to WiFi networks. In doing so, we identified several other performance barriers in WiFi networks, that we also offer solutions for. These include the 802.11 performance anomaly, the interaction between WiFi networks and latency-reducing measures for TCP, and the lack of a mechanism for airtime policy enforcement in WiFi networks.

Turning to a more general, but related, performance barrier in software-based network processing, we have investigated how the overhead that the operating system imposes on high-performance packet processing applications can be reduced. We have contributed a description and thorough evaluation of the Linux community's answer to this problem, which allows integration of packet processing programs into the networking stack in a cooperative way, achieving both flexibility and high performance.

Finally, over the course of implementing our solutions for the research problems, we have implemented a measurement tool designed to make experimental evaluations of networks more reliable and easier to perform. This tool is already widely used in the bufferbloat community, and can significantly increase the quality and availability of sophisticated network measurements in the future.

Looking forward, while the contributions presented here represent a significant step forward, there are certainly still performance barriers that prevent real-world networks from achieving their maximum performance potential. In the bufferbloat space, the most significant road-block is no doubt the effort needed to get the new technology deployed to the hundreds of millions of existing devices all over the world, many of which have an upgrade cycle measured in years. However, it is encouraging to see that awareness of the problem is at least increasing, with prominent examples being deployment in OS X,[2] and the inclusion of a bufferbloat measure in more online speed tests.[3]

As far as WiFi performance is concerned, there are several areas that warrant further study. Adopting the solutions presented here to the case

[2] `https://lists.bufferbloat.net/pipermail/bloat/2018-September/008615.html`

[3] E.g., `https://www.dslreports.com/speedtest` and `https://fast.com`.

where more functionality is offloaded into hardware is one such issue; either solutions have to be implemented in the hardware or firmware itself, or the operating system-level solutions need to take potential queueing at lower layers into account. Simultaneous transmission to multiple devices will likely play an increasing role in future WiFi deployments, which means scheduling algorithms need to be adapted to deal with the case where several queues need to be serviced simultaneously. In addition, re-evaluating the 802.11e QoS mechanism in the light of modern queueing mechanisms could be another interesting area of inquiry. Finally, while the solutions we have seen so far has been limited to single access point cases, it is quite probable that better results could be achieved by cooperation between neighbouring access points in a dense deployment scenario (which is becoming increasingly common as WiFi becomes ever more ubiquitous).

The high-performance software packet processing space is perhaps the most active of the subject areas we touch upon in this thesis. As we have seen, there are several competing architectures being explored; and future work includes adding features such as transport acceleration and full or partial hardware offload. Finally, there are no doubt many application areas that will benefit from the programmable capabilities enabled by frameworks such as XDP in the future.

In conclusion, the work presented here represents a significant contribution to improving the performance of real-world networks today and in the future. But it is clear that future improvements will continue to push the performance envelope towards ever higher capacity, lower latency and ubiquitous connectivity everywhere.

References

[1] J. Gettys and K. Nichols, "Bufferbloat: Dark buffers in the internet," *ACM Queue*, vol. 9, no. 11, pp. 40–54, Nov. 2011.

[2] C. Staff, "BufferBloat: what's wrong with the internet?" *Communications of the ACM*, vol. 55, no. 2, pp. 40–47, Feb. 2012.

[3] C. Kreibich *et al.*, "Netalyzr: illuminating the edge network," in *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*. ACM, 2010, pp. 246–259.

[4] S. Sundaresan *et al.*, "Broadband internet performance: a view from the gateway," in *ACM SIGCOMM computer communication review*, vol. 41. ACM, 2011, pp. 134–145.

[5] M. Chetty *et al.*, "Measuring broadband performance in South Africa," in *4th Annual Symposium on Computing for Development*. ACM, 2013.

[6] M. Dischinger *et al.*, "Characterizing residential broadband networks," in *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*. ACM, 2007, pp. 43–56.

[7] C. Chirichella and D. Rossi, "To the moon and back: are internet bufferbloat delays really that large?" in *2013 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. IEEE, 2013, pp. 417–422.

[8] H. Jiang *et al.*, "Tackling bufferbloat in 3g/4g networks," in *Proceedings of the 2012 Internet Measurement Conference*. ACM, 2012, pp. 329–342.

[9] S. Alfredsson *et al.*, "Impact of TCP congestion control on bufferbloat in cellular networks," in *IEEE 14th International Symposium and Workshops on a World of Wireless, Mobile and Multimedia Networks (WoWMoM)*, 2013.

[10] J. Aikat *et al.*, "Variability in TCP round-trip times," in *ACM conference on Internet measurement*. ACM, 2003, pp. 279–284.

[11] M. Allman, "Comments on bufferbloat," *ACM SIGCOMM Computer Communications Review*, vol. 43, no. 1, pp. 31–37, January 2013.

[12] S. Floyd and V. Jacobson, "Random early detection gateways for congestion avoidance," *IEEE/ACM Transactions on Networking*, vol. 1, no. 4, pp. 397–413, 1993.

[13] S. Floyd, R. Gummadi, and S. Shenker, "Adaptive RED: An algorithm for increasing the robustness of RED's active queue management," 2001. http://www.icir.org/floyd/papers.html

[14] P. McKenney, "Stochastic fairness queueing," in *INFOCOM '90. Ninth Annual Joint Conference of the IEEE Computer and Communication Societies*, vol. 2. IEEE, jun 1990, pp. 733–740.

[15] R. Adams, "Active Queue Management: A Survey," *IEEE Communications Surveys & Tutorials*, vol. 15, no. 3, pp. 1425–1476, 2013.

[16] M. May *et al.*, "Reasons not to deploy RED," in *1999 Seventh International Workshop on Quality of Service (IWQoS '99)*, 1999, pp. 260–262.

[17] K. Nichols and V. Jacobson, "Controlling queue delay," *Communications of the ACM*, vol. 55, no. 7, pp. 42–50, Jul. 2012.

[18] R. Pan *et al.*, "PIE: A lightweight control scheme to address the bufferbloat problem," in *2013 IEEE 14th International Conference on High Performance Switching and Routing (HPSR)*, July 2013, pp. 148–155.

[19] T. Høiland-Jørgensen *et al.*, "The Flow Queue CoDel Packet Scheduler and Active Queue Management Algorithm," RFC 8290 (Experimental), RFC Editor, Jan. 2018.

[20] G. Carofiglio and L. Muscariello, "On the impact of TCP and per-flow scheduling on internet performance," *IEEE/ACM Transactions on Networking*, vol. 20, no. 2, pp. 620–633, 2012.

[21] F. Checconi, L. Rizzo, and P. Valente, "QFQ: Efficient packet scheduling with tight guarantees," *IEEE/ACM Transactions on Networking (TON)*, vol. 21, no. 3, pp. 802–816, 2013.

[22] A. Kortebi *et al.*, "Evaluating the number of active flows in a scheduler realizing fair statistical bandwidth sharing," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 33. ACM, 2005, pp. 217–228.

[23] J. Corbet, "Network transmit queue limits," LWN Article, August 2011. https://lwn.net/Articles/454390/

[24] G. Niestegge, "The 'leaky bucket' policing method in the ATM (Asynchronous Transfer Mode) network," *International Journal of Communication Systems*, vol. 3, no. 2, pp. 187–197, 1990.

[25] A. Eleftheriadis and D. Anastassiou, "Constrained and general dynamic rate shaping of compressed digital video," in *International Conference on Image Processing*, vol. 3. IEEE, 1995, pp. 396–399.

[26] "IEEE standard for information technology–telecommunications and information exchange between systems local and metropolitan area networks–specific requirements part 11: Wireless LAN medium access control (MAC) and physical layer (PHY) specifications," *IEEE Std 802.11-2012 (Revision of IEEE Std 802.11-2007)*, March 2012.

[27] M. Heusse *et al.*, "Performance anomaly of 802.11 b," in *INFOCOM 2003. Twenty-Second Annual Joint Conference of the IEEE Computer and Communications. IEEE Societies*, vol. 2. IEEE, 2003, pp. 836–843.

[28] N. Khademi, D. Ros, and M. Welzl, "The new AQM kids on the block: Much ado about nothing?" Oslo University, Tech. Rep. 434, 2013.

[29] A. Showail, K. Jamshaid, and B. Shihada, "Buffer sizing in wireless networks: challenges, solutions, and opportunities," *IEEE Communications Magazine*, vol. 54, no. 4, pp. 130–137, Apr 2016.

[30] K. Cai *et al.*, "Wireless Unfairness: Alleviate MAC Congestion First!" in *Proceedings of the Second ACM International Workshop on Wireless Network Testbeds, Experimental Evaluation and Characterization*, ser. WinTECH '07. ACM, 2007, pp. 43–50.

[31] L. B. Jiang and S. C. Liew, "Proportional fairness in wireless LANs and ad hoc networks," in *Wireless Communications and Networking Conference, 2005 IEEE*, vol. 3. IEEE, 2005, pp. 1551–1556.

[32] P. Lin, W.-I. Chou, and T. Lin, "Achieving airtime fairness of delay-sensitive applications in multirate IEEE 802.11 wireless LANs," *IEEE Communications Magazine*, vol. 49, no. 9, pp. 169–175, 2011.

[33] L. Sanabria-Russo *et al.*, "Future evolution of CSMA protocols for the IEEE 802.11 standard," in *2013 IEEE International Conference on Communications Workshops (ICC)*. IEEE, 2013, pp. 1274–1279.

[34] T. Joshi *et al.*, "Airtime fairness for IEEE 802.11 multirate networks," *IEEE Transactions on Mobile Computing*, vol. 7, no. 4, pp. 513–527, Apr 2008.

[35] J. Dunn *et al.*, "A practical cross-layer mechanism for fairness in 802.11 networks," in *First International Conference on Broadband Networks (BroadNets 2004)*. IEEE, 2004, pp. 355–364.

[36] T. Razafindralambo *et al.*, "Dynamic packet aggregation to solve performance anomaly in 802.11 wireless networks," in *Proceedings of the 9th ACM international symposium on Modeling analysis and simulation of wireless and mobile systems*. ACM, 2006, pp. 247–254.

[37] M. Kim, E.-C. Park, and C.-H. Choi, "Adaptive two-level frame aggregation for fairness and efficiency in IEEE 802.11n wireless LANs," *Mobile Information Systems*, 2015.

[38] R. G. Garroppo *et al.*, "Providing air-time usage fairness in IEEE 802.11 networks with the deficit transmission time (DTT) scheduler," *Wireless Networks*, vol. 13, no. 4, pp. 481–495, Aug 2007.

[39] R. Riggio, D. Miorandi, and I. Chlamtac, "Airtime Deficit Round Robin (ADRR) packet scheduling algorithm," in *2008 5th IEEE International Conference on Mobile Ad Hoc and Sensor Systems*, Sep. 2008, pp. 647–652.

[40] M. Richart *et al.*, "Resource allocation for network slicing in WiFi access points," in *13th International Conference on Network and Service Management, CNSM, 2017*, 2017.

[41] K. Katsalis *et al.*, "Virtual 802.11 wireless networks with guaranteed throughout sharing," in *2015 IEEE Symposium on Computers and Communication (ISCC)*, Jul 2015.

[42] Y. Yiakoumis *et al.*, "Slicing home networks," in *Proceedings of the 2Nd ACM SIGCOMM Workshop on Home Networks*, ser. HomeNets '11. ACM, 2011.

[43] "Air Time Fairness (ATF) Phase1 and Phase 2 Deployment Guide," Cisco systems, 2015. https://www.cisco.com/c/en/us/td/docs/wireless/technology/mesh/8-2/b_Air_Time_Fairness_Phase1_and_Phase2_Deployment_Guide.html

[44] J. W. Lockwood *et al.*, "NetFPGA–an open platform for gigabit-rate network switching and routing," in *IEEE International Conference on Microelectronic Systems Education*. IEEE, 2007.

[45] P. Bosshart *et al.*, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, 2014.

[46] L. Rizzo and G. Lettieri, "Vale, a switched ethernet for virtual machines," in *Proceedings of the 8th international conference on Emerging networking experiments and technologies*. ACM, 2012.

[47] S. Han *et al.*, "PacketShader: a GPU-accelerated software router," in *ACM SIGCOMM Computer Communication Review*, vol. 40, no. 4. ACM, 2010.

[48] D. Kirchner *et al.*, "Augustus: a CCN router for programmable networks," in *Proceedings of the 3rd ACM Conference on Information-Centric Networking*. ACM, 2016.

[49] P. M. Santiago del Rio *et al.*, "Wire-speed statistical classification of network traffic on commodity hardware," in *Proceedings of the 2012 Internet Measurement Conference*. ACM, 2012.

[50] R. B. Mansilha *et al.*, "Hierarchical content stores in high-speed ICN routers: Emulation and prototype implementation," in *Proceedings of the 2nd ACM Conference on Information-Centric Networking*. ACM, 2015.

[51] P. Emmerich *et al.*, "Moongen: A scriptable high-speed packet generator," in *Proceedings of the 2015 Internet Measurement Conference*. ACM, 2015.

[52] S. Han *et al.*, "MegaPipe: A new programming interface for scalable network I/O," in *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI '12)*, 2012.

[53] T. Marian, K. S. Lee, and H. Weatherspoon, "NetSlices: scalable multi-core packet processing in user-space," in *Proceedings of the eighth ACM/IEEE symposium on Architectures for networking and communications systems*. ACM, 2012.

[54] L. Linguaglossa *et al.*, "High-speed software data plane via vectorized packet processing," Telecom ParisTech, Tech. Rep., 2017.

[55] R. Morris *et al.*, "The Click modular router," *ACM SIGOPS Operating Systems Review*, vol. 33, no. 5, 1999.

[56] M. Dobrescu *et al.*, "RouteBricks: exploiting parallelism to scale software routers," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 2009.

[57] B. Pfaff *et al.*, "The design and implementation of Open vSwitch," in *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI '15)*, 2015.

[58] L. Deri, "Modern packet capture and analysis: Multi-core, multi-gigabit, and beyond," in *the 11th IFIP/IEEE International Symposium on Integrated Network Management (IM)*, 2009.

[59] L. Rizzo, "Netmap: a novel framework for fast packet I/O," in *21st USENIX Security Symposium (USENIX Security 12)*, 2012.

[60] S. Peter *et al.*, "Arrakis: The operating system is the control plane," *ACM Transactions on Computer Systems (TOCS)*, vol. 33, no. 4, 2016.

[61] J. Martins *et al.*, "ClickOS and the art of network function virtualization," in *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*. USENIX Association, 2014.

[62] "PF_RING ZC (Zero Copy)," Ntop project, 2018. https://www.ntop.org/products/packet-capture/pf_ring/pf_ring-zc-zero-copy/

[63] "OpenOnload," Solarflare Communications Inc, 2018. https://www.openonload.org/

[64] "Data plane development kit," Linux Foundation, 2018. https://www.dpdk.org/

[65] M. Tedre, *The science of computing: Shaping a discipline*. CRC Press, 2014.

[66] H. Bidgoli, *The Internet Encyclopedia, Volume 2 (G - O)*. Wiley, 2004.

Paper I

Measuring Latency Variation in the Internet

Reprinted from

ACM CoNEXT '16, December 12–15, 2016, Irvine, CA, USA

"Science cannot progress without reliable and accurate
measurement of what it is you are trying to study.
The key is measurement, simple as that."

Robert D. Hare

Measuring Latency Variation in the Internet

Toke Høiland-Jørgensen, Bengt Ahlgren, Per Hurtig and Anna Brunstrom

toke.hoiland-jorgensen@kau.se, bengta@sics.se, per.hurtig@kau.se, anna.brunstrom@kau.se

Abstract

We analyse two complementary datasets to quantify the latency variation experienced by internet end-users: (i) a large-scale active measurement dataset (from the Measurement Lab Network Diagnostic Tool) which shed light on long-term trends and regional differences; and (ii) passive measurement data from an access aggregation link which is used to analyse the edge links closest to the user.

The analysis shows that variation in latency is both common and of significant magnitude, with two thirds of samples exceeding 100 ms of variation. The variation is seen within single connections as well as between connections to the same client. The distribution of experienced latency variation is heavy-tailed, with the most affected clients seeing an order of magnitude larger variation than the least affected. In addition, there are large differences between regions, both within and between continents. Despite consistent improvements in throughput, most regions show no reduction in latency variation over time, and in one region it even increases.

We examine load-induced queueing latency as a possible cause for the variation in latency and find that both datasets readily exhibit symptoms of queueing latency correlated with network load. Additionally, when this queueing latency does occur, it is of significant magnitude, more than 200 ms in the median. This indicates that load-induced queueing contributes significantly to the overall latency variation.

1 Introduction

As applications turn ever more interactive, network latency plays an increasingly important role for their performance. The end-goal is to get as close as possible to the physical limitations of the speed of light [1]. However, today the latency of internet connections is often larger than it needs to be. In this

work we set out to quantify how much. Having this information available is important to guide work that sets out to improve the latency behaviour of the internet; and for authors of latency-sensitive applications (such as Voice over IP, or even many web applications) that seek to predict the performance they can expect from the network.

Many sources of added latency can be highly variable in nature. This means that we can quantify undesired latency by looking specifically at the *latency variation* experienced by a client. We do this by measuring how much client latency varies above the minimum seen for that client. Our analysis is based on two complementary sources of data: we combine the extensive publicly available dataset from the Measurement Lab Network Diagnostic Tool (NDT) with a packet capture from within a service provider access network. The NDT data, gathered from 2010 to 2015, comprises a total of 265.8 million active test measurements from all over the world. This allows us to examine the development in latency variation over time and to look at regional differences. The access network dataset is significantly smaller, but the network characteristics are known with greater certainty. Thus, we can be more confident when interpreting the results from the latter dataset. These differences between the datasets make them complement each other nicely.

We find that significant latency variation is common in both datasets. This is the case both within single connections and between different connections from the same client. In the NDT dataset, we also observe that the magnitude of latency variation differs between geographic regions, both between and within continents. Looking at the development over time (also in the NDT dataset), we see very little change in the numbers. This is in contrast to the overall throughput that has improved significantly.

One important aspect of latency variation is the correlation between increased latency and high link utilisation. Queueing delay, in particular, can accumulate quickly when the link capacity is exhausted, and paying attention to such scenarios can give insight into issues that can cause real, if intermittent, performance problems for users. We examine queueing delay as a possible source of the observed latency variation for both datasets, and find strong indications that it is present in a number of instances. Furthermore, when queueing latency does occur it is of significant magnitude.

The rest of the paper is structured as follows: Section 2 introduces the datasets and the methodology we have used to analyse them. Section 3 discusses the large-scale variations in latency over time and geography, and Section 4 examines delay variation on access links. Section 5 presents our examination of load-induced queueing delay. Finally, Section 6 discusses related work and Section 7 concludes the paper.

2 Datasets and methodology

The datasets underlying our analysis are the publicly available dataset from Measurement Lab (M-Lab), specifically the Network Diagnostic Tool (NDT) data [2], combined with packet header traces from access aggregation links of

Table 1: Total tests per region and year (millions).

Reg.	2010	2011	2012	2013	2014	2015
AF	0.65	0.63	0.79	0.72	0.76	0.57
AS	7.79	7.45	6.55	5.75	5.83	4.57
EU	35.00	31.12	27.96	22.40	21.23	16.82
NA	11.70	8.51	7.90	7.01	7.06	8.00
SA	2.68	1.73	2.83	2.94	2.05	1.26
OC	1.33	1.33	0.79	0.55	0.65	0.57
Total	59.22	50.81	46.87	39.43	37.60	31.79

an internet service provider. This section presents each of the datasets, and the methodology we used for analysis.

2.1 The M-Lab NDT data

The M-Lab NDT is run by users to test their internet connections. We use the 10-second bulk transfer from the server to the client, which is part of the test suite. When the test is run, the client attempts to pick the nearest server from the geographically distributed network of servers provided by the M-Lab platform. The M-Lab servers are globally distributed[4], although with varying density in different regions.

The server is instrumented with the Web100 TCP kernel instrumentation [3], and captures several variables of the TCP state machine every 5 ms of the test. Data is available from early 2009, and we focus on the six year period 2010–2015, comprising a total of 265.8 million test runs. Table 1 shows the distribution of test runs for the years and regions we have included in our analysis.

Since the NDT is an active test, the gathered data is not a representative sample of the traffic mix flowing through the internet. Instead, it may tell us something about the *links* being traversed by the measurement flows. Looking at links under load is interesting, because some important effects can be exposed in this way, most notably bufferbloat: Loading up the link causes any latent buffers to fill, adding latency that might not be visible if the link is lightly loaded. This also means that the baseline link utilisation (e.g., caused by diurnal usage patterns) become less important: an already loaded link can, at worst, result in a potentially higher baseline latency. This means that the results may be biased towards showing lower latency variation than is actually seen on the link over time. But since we are interested in establishing a lower bound on the variation, this is acceptable.

For the base analysis, we only exclude tests that were truncated (had a total run time less than 9 seconds). We use the TCP RTT samples as our data

[4]See `https://www.measurementlab.net/status/` for more information on the server placements.

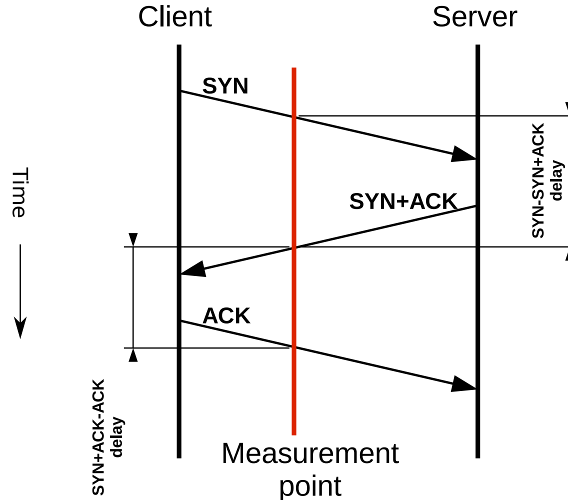


Figure 1: Delays computed from the TCP connection setup.

points, i.e., the samples computed by the server TCP stack according to Karn's algorithm [4], and focus on the *RTT span*, defined as the difference between the minimum and maximum RTT observed during a test run. However, we also examine a subset of the data to assess what impact the choice of min and max observed RTT has on the data when compared to using other percentiles for each flow (see Section 3.3).

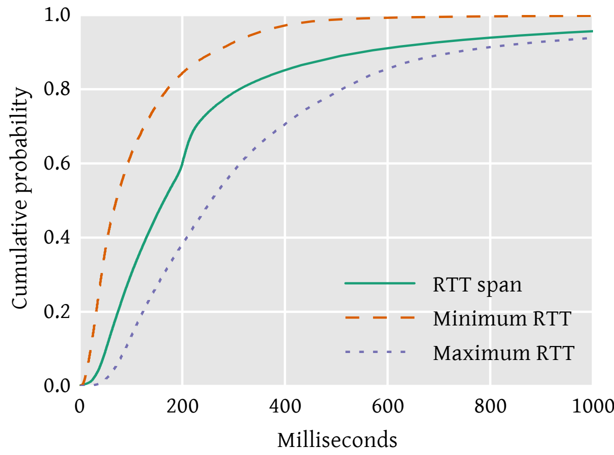
2.2 The access aggregation link data

The second dataset comes from two access aggregation links of an internet service provider. The links aggregate the traffic for about 50 and 400 clients, respectively, and connect them to the core network of the service provider. The traffic was captured passively at different times distributed over an eight-month period starting at the end of 2014. The average loads on the 1 Gbps links were, respectively, about 200 and 400 Mbit/s during peak hours. This dataset is an example of real internet traffic, since we are not generating any traffic.

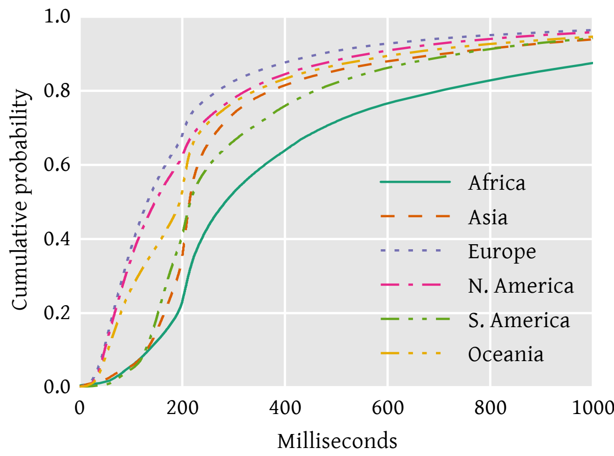
We analyse the delay experienced by the TCP connection setup packets in this dataset. The TCP connection setup consists of a three-way handshake with SYN, SYN+ACK, and ACK packets, as illustrated in Figure 1.

For the purpose of this paper, we study the client side of connections made to the public internet. That is, we study the path from the client, over the access link, to the measured aggregation link. This allows us to examine increased delays due to excess queuing in consumer equipment, and ensures that the path we measure is of known length.

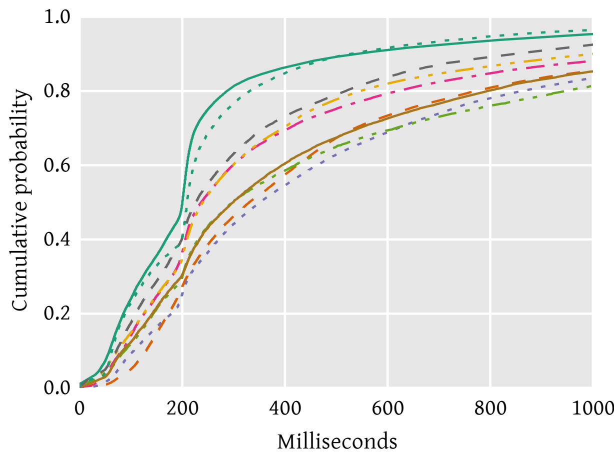
We examine the data for outgoing connections, i.e., connections that are initiated from the access side and connect to the public internet, which means we compute the round-trip delay between the SYN+ACK packet and the first ACK packet in the three-way handshake. The variation in these delay values is likely to be caused by queueing, since connection endpoints normally



(a)



(b)



(c)

Figure 2: RTT values computed over individual flows. (a) Min and max RTT and the span between them, for all flows. (b) Distribution of per-flow RTT span per continent (2015 data). (c) Distribution of per-flow RTT span per country in Africa (2015 data for countries with $n > 10,000$).

respond immediately. We also compute the instantaneous load at each sample and examine the correlation between delay and load for a few clients.

2.3 Sources of latency variation

Naturally, the observed latency variation can have several causes [5]. These include queueing delay along the path, delayed acknowledgements, transmission delay, media access delays, error recovery, paths changing during the test and processing delays at end-hosts and intermediate nodes. For the main part of our analysis, we make no attempt to distinguish between different causes of latency variation. However, we note that latency variation represents latency that is superfluous in the sense that it is higher than the known attainable minimum for the path. In addition, we analyse a subset of each dataset to examine to what extent queueing latency is a factor in the observed latency variation.

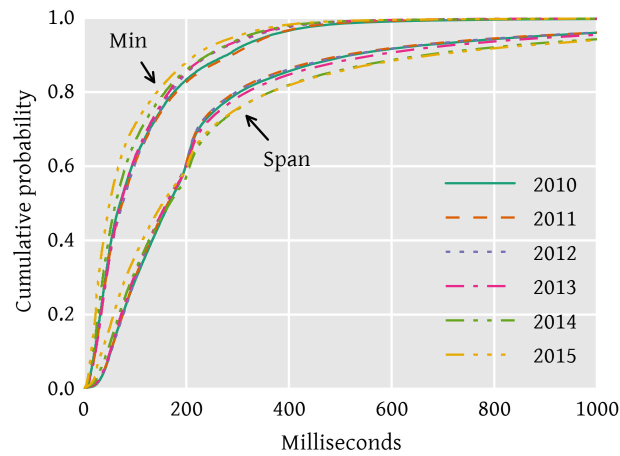
We believe that the chosen datasets complement each other nicely and allow us to illuminate the subject from different angles, drawing on the strengths of them both. The NDT dataset, being based on active measurements, allows us to examine connections that are being deliberately loaded, and the size of the dataset allows us to examine temporal and geographic trends. The access network dataset, on the other hand, has smaller scope but the examined path is known; and so we can rule out several sources of delay and be more confident when interpreting the results.

3 Latency variation over time and geography

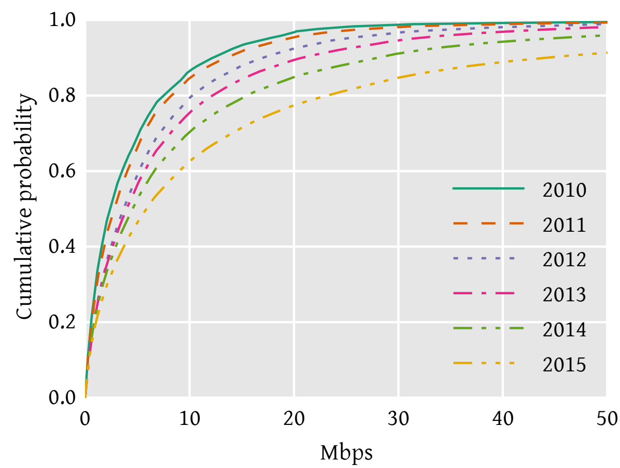
In this section, we analyse the M-Lab NDT dataset to explore geographic differences, and also look at the development of the RTT span over time. For an initial overview, Figure 2a shows the distribution of the RTT span in the whole M-Lab NDT dataset, along with the minimum and maximum RTTs it is derived from. This shows a significant amount of extra latency: two thirds of samples exceed 100 ms of RTT span, with the 95th percentile exceeding 900 ms.

3.1 Geographic differences

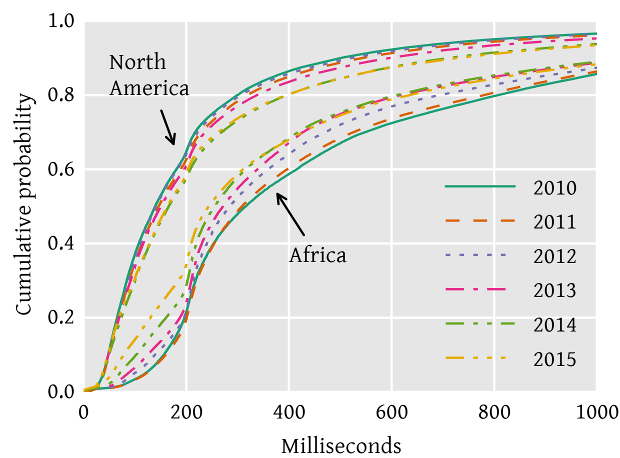
Figure 2b shows the RTT span distributed geographically per continent for the 2015 data. This shows a significant difference between regions, with the median differing by more than a factor of two between the best and the worst region. Looking within these regions, Figure 2c shows the per-country distributions within Africa. Here, the heavy tail of latencies above one second affects as much as 20% of the samples from the country with the highest latency. These high latencies are consistent with previous studies of African internet connections [6]. The data for Europe (omitted due to space constraints) shows that the difference among European countries is of the same magnitude as the difference among continents.



(a)



(b)



(c)

Figure 3: Development over time of per-flow latency and throughput in the NDT dataset. (a) Min RTT and RTT span, per year. (b) Per-flow average throughput, per year. (c) The development in latency span over time for North America and Africa.

3.2 Development over time

Figure 3a shows the minimum RTT and the RTT span for each of the years in the dataset. While the minimum RTT has decreased slightly over the years, no such development is visible for the span. This is striking when compared to the development in throughput, which has increased consistently, as shown in Figure 3b. Some of this increase in average throughput may be due to other factors than increase in link capacity (e.g. protocol efficiency improvements). Even so, the disparity is clear: while throughput has increased, RTT span has not decreased, and the decrease in minimum RTT is slight.

Looking at this development for different continents, as shown in Figure 3c, an increase in latency span over the years is seen in North America while Africa has seen a consistent, but small, reduction in latency span over time. This latter development is most likely due to developments in infrastructure causing traffic to travel fewer hops, thus decreasing the potential sources of extra latency.

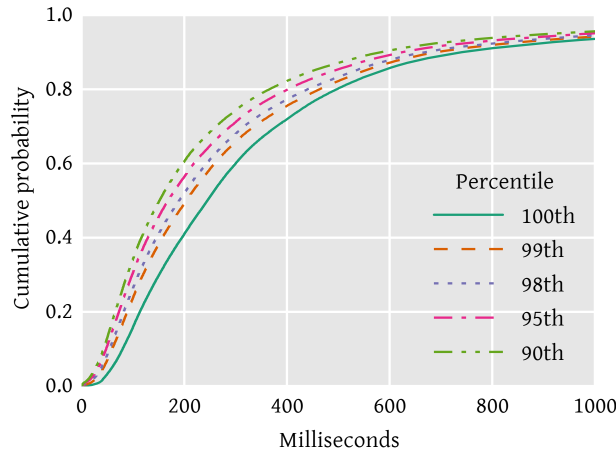
3.3 Different measures of latency span

The way the NDT dataset is structured makes the per-flow min and max RTT values the only ones that are practical to analyse for the whole dataset. To assess what effect this choice of metric has on the results, we performed a more detailed analysis for a subset of the data. Figure 4a shows the latency span distribution for the data from August 2013 when using percentiles of the per-flow RTT measurements ranging from 90 to 99 in place of the max. We see that in this case the median measured RTT span drops to between 151 ms and 204 ms, from 250 ms when using the max — a drop of between 17% and 40%. It is not clear that the max is simply an outlier for all flows; but for those where it is, our results will overestimate the absolute magnitude of the RTT span. However, the shape of the distribution stays fairly constant, and using the max simply leads to higher absolute numbers. This means that we can still say something about trends, even for those flows where the max RTT should be considered an outlier.

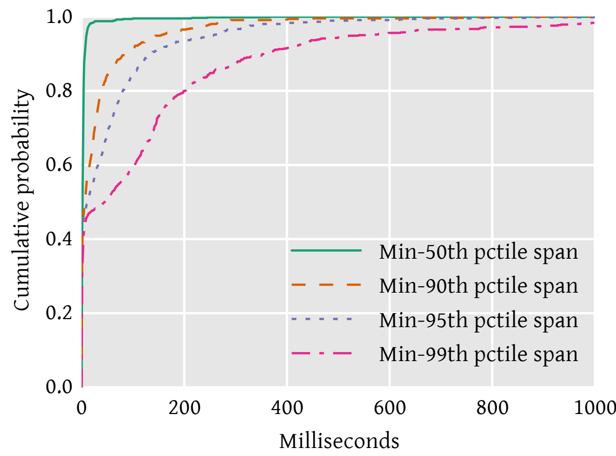
4 Latency variation in the access network

In this section we analyse the latency variation of TCP 3-way handshakes in the access network dataset. Figure 4b shows the distribution of the per-client RTT variation, computed as the span between the per-client minimum delay and the respective percentiles of samples to that client. To ensure that we do not mistakenly use a too low minimum delay value, only handshakes which did not have any SYN+ACK retransmissions are considered when computing the minimum.

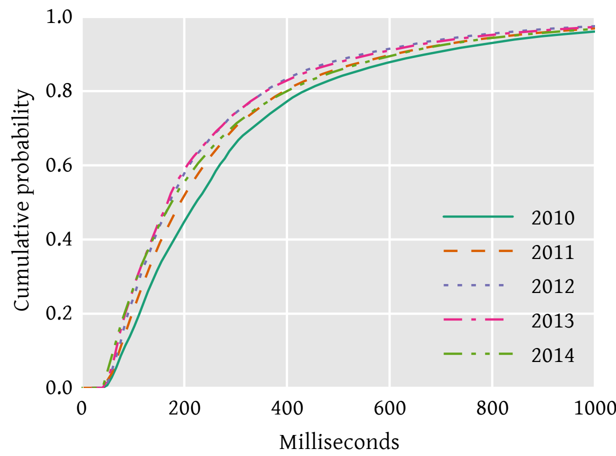
For about half of the client population covered by the link shown in Figure 4b, delay increases substantially over the minimum at times. For example, 20% of the clients experience increased delays of more than about



(a)



(b)



(c)

Figure 4: (a) Latency span for all flows in August 2013 when using different percentiles to determine the max RTT. NDT dataset. (b) Client side round trip delay percentiles over all clients relative to the minimum delay. A full day of the first aggregation link in the access network dataset. (c) Distribution of the magnitude of detected queueing delay, per year. Flows with detected queueing latency, NDT dataset.

80 ms, at least 5% of the time. The second link shows similar behaviour, but has *more* clients that are affected by increased delay.

Another interesting feature of the data is that there is a significant difference in the magnitude of the latency span depending on which percentile of latency measurements one looks at. That is, if we consider the per-user 99th percentile of latency rather than the 95th, suddenly more than half the users experience latency variations in excess of 100 ms for the first link, and more than 80% for the second link. This underscores the fact that delay spikes can be a very transient problem, but one that is of significant magnitude when it does occur.

Comparing with the NDT dataset, the analysis of the access link data shows a lower frequency of latency variation, as well as a lower magnitude of the variation when it does occur. However, both datasets show that significant latency variation occurs for a considerable fraction of users. We attribute the difference in magnitude to the difference in measurement methods: the NDT measurements are taken while the link is deliberately loaded, while not all measurements from the access network are taken from saturated links.

5 Examining queueing latency

As mentioned in Section 2.3, latency variations can have many causes, and without having insight into the network path itself it can be difficult to identify which are the most prevalent. However, experience from more controlled environments (such as experiments performed to evaluate AQM algorithms [7]) suggests that queueing delay can be a significant source. Due to the magnitude of the variation we see here, we conjecture that this is also the case in this dataset. To examine this further, in this section we present an analysis of the queueing delay of a subset of the traffic in both datasets. We aim to perform a conservative analysis, and so limit ourselves to tests for which it is possible to identify queueing latency with high certainty.

5.1 Latency reductions after a drop

Our analysis is based upon a distinct pattern, where the sample RTT increases from the start of a flow until a congestion event, then sharply decreases afterwards. An example of this pattern is seen in Figure 5. This pattern is due to the behaviour of TCP: The congestion control algorithm will increase its sending rate until a congestion event occurs, then halve it. If a lot of packets are queued when this happens, the queue has a chance to drain, and so subsequent RTT samples will show a lower queueing delay. Thus, it is reasonable to assume that when this pattern occurs, the drop in RTT is because the queue induced by the flow dissipates as it slows down. So when we detect this sharp correlation between a congestion event and a subsequent drop in RTT, we can measure the magnitude of the drop and use it as a lower bound on queueing delay.

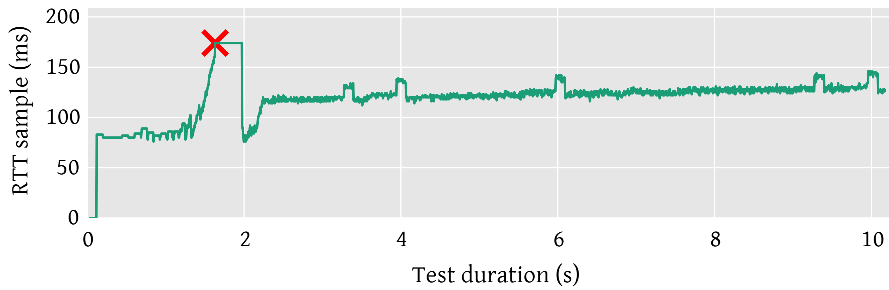


Figure 5: Example of the drop in RTT after a congestion event. The red cross marks the congestion event.

We limit the analysis to flows that have exactly one congestion event, and spend most of its lifetime being limited by the congestion window. Additionally, we exclude flows that are truncated or transfer less than 0.2 MB of data. For the remaining flows, we identify the pattern mentioned above by the following algorithm:

1. Find three values: *first_rtt*, the first non-zero RTT sample; *cong_rtt*, the RTT sample at the congestion event; and *cong_rtt_next*, the first RTT sample after the event that is different from *cong_rtt*.

2. Compute the differences between *first_rtt* and *cong_rtt* and between *cong_rtt* and *cong_rtt_next*. If both of these values are above 40 ms,[5] return the difference between *cong_rtt* and *cong_rtt_next*.

We add a few minor refinements to increase the accuracy of the basic algorithm above:[6]

1. When comparing *first_rtt* and *cong_rtt*, use the median of *cong_rtt* and the two previous RTT samples. This weeds out tests where only a single RTT sample (coinciding with the congestion event) is higher than the baseline.

2. When comparing *cong_rtt* and *cong_rtt_next*, use the minimum of the five measurements immediately following *cong_rtt_next*. This makes sure we include cases where the decrease after the congestion event is not instant, but happens over a couple of RTT samples.

3. Compute the maximum span between the largest and smallest RTT sample in a sliding window of 10 data samples over the time period following the point of *cong_rtt_next*. If this span is higher than the drop in RTT after the congestion event, filter out the flow.

[5]The threshold is needed to exclude naturally occurring variation in RTT samples from the detection. We found 40 ms empirically to be a suitable conservative threshold: It is the lowest value that did not result in a significant number of false positives.

[6]The full code and dataset is published at `https://www.cs.kau.se/tohojo/measuring-latency-variation/`

By applying the algorithm to the data from 2010 through 2014[7], we identified a total of 5.7 million instances of the RTT pattern, corresponding to 2.4% of the total number of flows. While this is a relatively small fraction of the flows, in this section we have aimed to be conservative and only pick out flows where we can algorithmically identify the source of the extra latency as queueing delay with a high certainty. This does not mean, however, that queueing delay cannot also be a source of latency variation for other flows.

Figure 4c shows the distribution of the magnitude of the detected queueing delay. We see that this follows a similar heavy-tailed distribution as the total latency variation. In addition, of those tests that our algorithm identifies as experiencing self-induced queueing, a significant percentage see quite a lot of it: 80% is above 100 ms, and 20% is above 400 ms. We see a downward trend in the queueing delay magnitude from 2010 to 2012/13, with a slight increase in 2014.

Based on our analysis of this subset of the whole dataset, we conclude that (i) queueing delay is present in a non-trivial number of instances and that (ii) when it does occur, it is of significant magnitude.

5.2 Delay correlated with load

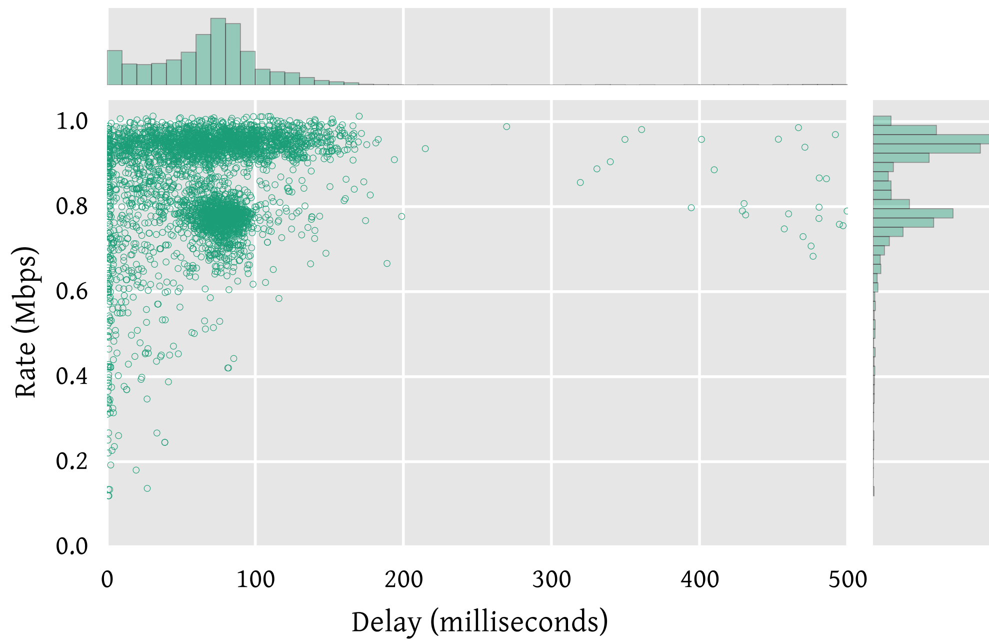
A network is more likely to exhibit queueing delay when it is congested. Thus, delay correlated with load can be an indication of the presence of queueing delay. When analysing the access network dataset, we identified several cases where strong correlation between delay and load existed. In this section we look at two examples of this behaviour.

Figures 6a and 6b show the correlation between client side delay and the instantaneous outbound load during the 200 ms just preceding each delay sample. The sample period is one hour during peak time (20.30–21.30).

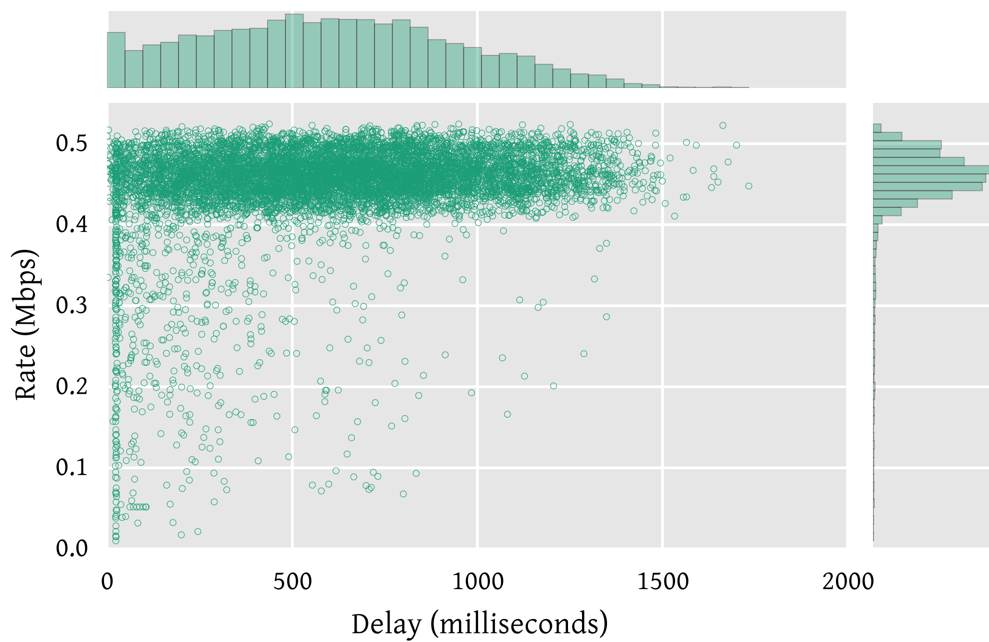
For the first client, we see two clusters of delay/load values, indicating two kinds of behaviour. There is one cluster just under a performance ceiling of 1 Mbit/s, but with increased round-trip delays up to almost 200 ms. This behaviour clearly indicates a saturated uplink where the upstream bottleneck limits the throughput and induces queueing latency. The other cluster is centred around about 0.8 Mbit/s and 80 ms increase in round-trip delay. This indicates an equilibrium where the outbound capacity is not the primary bottleneck, and so doesn't induce as much queueing delay. In addition to these clusters, there are some scattered data points up to just over 3 s increase in round-trip delay (not all of which are visible on the figure). The second client is clearly limited by a low uplink capacity, resulting in very large delays — up to about 1.5 s, which is consistent with the low capacity resulting in large queue drain times.

Together, the behaviour of these two client links (along with additional clients we have examined but not included here) clearly show that load-induced

[7] The Measurement Lab dataset was restructured in the middle of 2015, making it difficult to apply the detailed analysis for the 2015 data. For that reason, we have not included 2015 in these results.



(a)



(b)

Figure 6: Delay and instantaneous outbound rate for two clients in the access network dataset. The histograms on the axes show the marginal distributions for the rate and delay samples.

queueing delay is one source of the latency variation we have observed in the analysis of the whole dataset. According to the service provider, the access network otherwise does not have the amount of buffering needed for the delays we see in our measurements, pointing to large buffers in consumer equipment as the likely culprit.

5.3 Discussion

The analysis presented in this section indicates that excess queueing latency (i.e., bufferbloat) is indeed a very real and routinely occurring phenomenon. While we do not claim to have a means of accurately quantifying bufferbloat in all instances, we have sought to compensate for the lack of accuracy by erring on the side of caution in identifying bloat. And the fact that signs of bufferbloat so readily appears in both datasets constitutes a strong indicator that bufferbloat is indeed prevalent in real networks.

Another finding is that in the cases where bufferbloat does appear, it tends to be significant: most often on the order of several hundreds of milliseconds. This means that when bufferbloat does appear, it is quite noticeable and a considerable inconvenience for the end-user.

6 Related work

Several other studies have looked at the latency characteristics of internet traffic. These fall roughly into three categories: studies based on large-scale active measurements, studies based on targeted active measurements of a more limited scope, and passive measurements performed at various vantage points in the network. In this section we provide an overview of each of these categories in turn.

6.1 Large-scale active measurements

The *speedtest.net* measurement tool and the Netalyzr test suite are popular performance benchmark tools in wide use. Canadi et al [8] perform a study based on 54 million test runs from the former which shows very low baseline (unloaded) latencies, but considers neither latency variation nor development over time. Kreibich et al [9] base their study on 130,000 tests from the latter, and show queueing latency on the order of hundreds of milliseconds, but does not consider differences over time or between regions.

Another approach to large-scale active measurements is taken by the BISMark and SamKnows measurement platforms, both of which provide instrumented gateways to users. A study based on this data by Sundaresan et al [10] measures baseline and under-load latency and shows significant buffering in head-end equipment. Chetty et al [6] also use BISMark data (as well as other sources) to measure broadband performance in South Africa. Consistent with our results for this continent, they find that latencies are generally high, often on the order of several hundred milliseconds.

Another large-scale active measurement effort is the Dasu platform [11], which is a software client users install on their machines. This study does not focus on latency measurements, but it includes HTTP latency figures which indicate a large regional variation, not unlike what we observe.

Finally, the M-Lab Consortium has studied ISP interconnections [12] using the same data as we use. However, this study only considers aggregate latency over large time scales.

6.2 Targeted active measurements

Dischinger et al [13] and Choy et al [14], both use active probing of residential hosts to measure network connections. The former study finds that queueing delay in the broadband equipment is extensive, while the latter finds that a significant fraction of users experience too high latency to run games in the cloud. Bischof et al [15] perform a slightly different type of measurements, piggy-backing on the BitTorrent protocol, and find a majority of the users see median last-mile latency between 10 and 100 ms, with the 95th percentile of seeing several hundred milliseconds. A similar conclusion, but specifically targeted at assessing queueing latency, is reached in [16], which estimates that 10% of users experience a 90th percentile queueing latency above 100 ms.

Another type of targeted active measurements are performed by clients under the experimenters' control to examine the access network. These types of experiments are performed by, e.g., Jiang et al [17] and Alfredsson et al [18] to measure bufferbloat in cellular networks. Both studies find evidence of bufferbloat on the order of several hundred ms.

6.3 Passive measurements

Several studies perform passive measurements of backbone or other high-speed links between major sites [19–21]. They generally find fairly low and quite stable latencies, with the backbone link experiencing latencies dominated by the speed of light, and the others generally seeing median latencies well below 100 ms. Jaiswal et al [21] additionally measure RTT variations and find that the median variation is around 2–300 ms and the 95th percentile variation is on the order of several seconds. In addition, Pathak et al [22] perform passive measurement of latency inflation in MPLS overlay networks and find that inflation is common, mostly due to path changes in the underlying tunnels.

Another technique for passive measurements consists of taking captures at the edge of a network and analysing that traffic. Aikat et al [23] and Allman [24] both employ this technique to analyse the RTT of TCP connections between hosts inside and outside the network where the measurement is performed. Both studies analyse the latency variation, Aikat et al finding it to be somewhat higher than Allman.

Another vantage point for passive measurements is at edge networks. Such studies are performed by Vacirca et al [25] and Maier et al [26] for mobile and residential networks, respectively. The former study finds that RTT can vary

greatly over connections, while the latter finds that the baseline latency of the TCP handshake is dominated by the client part.

Finally, Hernandez-Campos and Papadopouli [27] compares wired and wireless traffic by means of passive packet captures. They find that wireless connections experience a much larger RTT variation than wired connections do.

7 Conclusions

We have analysed the latency variation experienced by clients on the internet by examining two complementary datasets from active measurement tests and from traffic captures from an ISP. In addition, we have analysed a subset of the data to attempt to determine whether load-induced queueing delay in the network can be part of the reason for the large variations. Based on our analysis, we conclude:

- Latency variation is both common and of significant magnitude, both within single connections and between connections to the same client. This indicates that it has a large potential to negatively affect end-user perceived performance.
- The worst affected clients see an order of magnitude larger variation than the least affected, and the choice of per-client percentile for the measured latency significantly affects the resulting conclusions. This indicates that latency spikes are transient and non-uniformly distributed.
- While throughput has increased over the six years we have examined, both minimum latency and latency variation have remained stable, and even increased slightly. This indicates that the improvements in performance afforded by the development of new technology are not improving latency.
- We find that in both datasets load-induced queueing delay is an important factor in latency variation, and quite significant in magnitude when it does occur. This indicates that load-induced queueing does in fact contribute significantly to the variation we see in overall latency behaviour.

8 Acknowledgements

We wish to thank Matt Mathis and Collin Anderson for suggesting we look at the Measurement Lab dataset, and for their helpful comments and technical expertise. We also wish to thank Henrik Abrahamsson for the use of his tools for computing instantaneous load from the access link data.

This work was in part funded by The Knowledge Foundation (KKS) through the SIDUS READY project.

References

[1] A. Singla *et al.*, “The internet at the speed of light,” in *13th ACM Workshop on Hot Topics in Networks*. ACM, 2014.

[2] “NDT test methodology,” 2015. https://github.com/ndt-project/ndt/wiki/NDTTestMethodology

[3] M. Mathis, J. Heffner, and R. Raghunarayan, “TCP Extended Statistics MIB,” RFC 4898 (Proposed Standard), Internet Engineering Task Force, May 2007.

[4] V. Paxson *et al.*, “Computing TCP’s Retransmission Timer,” RFC 6298 (Proposed Standard), Internet Engineering Task Force, Jun. 2011.

[5] B. Briscoe *et al.*, “Reducing internet latency: A survey of techniques and their merits,” *IEEE Communications Surveys Tutorials*, vol. 18, no. 99, pp. 2149–2196, 2014.

[6] M. Chetty *et al.*, “Measuring broadband performance in South Africa,” in *4th Annual Symposium on Computing for Development*. ACM, 2013.

[7] T. Høiland-Jørgensen, P. Hurtig, and A. Brunstrom, “The Good, the Bad and the WiFi: Modern AQMs in a residential setting,” *Computer Networks*, vol. 89, pp. 90–106, Oct. 2015.

[8] I. Canadi, P. Barford, and J. Sommers, “Revisiting broadband performance,” in *Proceedings of the 2012 Internet Measurement Conference*. ACM, 2012, pp. 273–286.

[9] C. Kreibich *et al.*, “Netalyzr: illuminating the edge network,” in *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*. ACM, 2010, pp. 246–259.

[10] S. Sundaresan *et al.*, “Broadband internet performance: a view from the gateway,” in *ACM SIGCOMM computer communication review*, vol. 41. ACM, 2011, pp. 134–145.

[11] M. A. Sánchez *et al.*, “Dasu: Pushing experiments to the internet’s edge.” in *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI ’13)*, 2013, pp. 487–499.

[12] M. Lab, “ISP interconnection and its impact on consumer internet performance,” Measurement Lab Consortium, Tech. Rep., October 2014.

[13] M. Dischinger *et al.*, “Characterizing residential broadband networks,” in *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*. ACM, 2007, pp. 43–56.

[14] S. Choy *et al.*, “The brewing storm in cloud gaming: A measurement study on cloud to end-user latency,” in *11th annual workshop on network and systems support for games*. IEEE Press, 2012, p. 2.

[15] Z. S. Bischof, J. S. Otto, and F. E. Bustamante, “Up, down and around the stack: ISP characterization from network intensive applications,” *ACM SIGCOMM Computer Communication Review*, vol. 42, no. 4, pp. 515–520, 2012.

[16] C. Chirichella and D. Rossi, “To the moon and back: are internet bufferbloat delays really that large?” in *2013 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. IEEE, 2013, pp. 417–422.

[17] H. Jiang *et al.*, “Tackling bufferbloat in 3g/4g networks,” in *Proceedings of the 2012 Internet Measurement Conference*. ACM, 2012, pp. 329–342.

[18] S. Alfredsson *et al.*, “Impact of TCP congestion control on bufferbloat in cellular networks,” in *IEEE 14th International Symposium and Workshops on a World of Wireless, Mobile and Multimedia Networks (WoWMoM)*, 2013.

[19] H. Jiang and C. Dovrolis, “Passive estimation of TCP round-trip times,” *ACM SIGCOMM Computer Communication Review*, vol. 32, no. 3, pp. 75–88, 2002.

[20] C. Fraleigh *et al.*, “Packet-level traffic measurements from the Sprint IP backbone,” *IEEE Network*, vol. 17, no. 6, pp. 6–16, 2003.

[21] S. Jaiswal *et al.*, “Inferring TCP connection characteristics through passive measurements,” in *INFOCOM 2004*, vol. 3. IEEE, 2004, pp. 1582–1592.

[22] A. Pathak *et al.*, “Latency inflation with MPLS-based traffic engineering,” in *ACM conference on Internet measurement*. ACM, 2011, pp. 463–472.

[23] J. Aikat *et al.*, “Variability in TCP round-trip times,” in *ACM conference on Internet measurement*. ACM, 2003, pp. 279–284.

[24] M. Allman, “Comments on bufferbloat,” *ACM SIGCOMM Computer Communications Review*, vol. 43, no. 1, pp. 31–37, January 2013.

[25] F. Vacirca, F. Ricciato, and R. Pilz, “Large-scale RTT measurements from an operational UMTS/GPRS network,” in *First International Conference on Wireless Internet*. IEEE, 2005.

[26] G. Maier *et al.*, “On dominant characteristics of residential broadband internet traffic,” in *ACM conference on Internet measurement*. ACM, 2009, pp. 90–102.

[27] F. Hernandez-Campos and M. Papadopouli, “Assessing The Real Impact of 802.11 WLANs: A Large-Scale Comparison of Wired and WirelessTraffic,” in *14th IEEE Workshop on Local and Metropolitan Area Networks*. IEEE, 2005.

Paper II

The Good, the Bad and the WiFi

Modern AQMs in a Residential Setting

Reprinted from

Computer Networks, vol 89, pg 90–106, October 2015

"I'm British; I know how to queue."

Arthur Dent, The Hitchhiker's Guide to the Galaxy

The Good, the Bad and the WiFi

Modern AQMs in a Residential Setting

Toke Høiland-Jørgensen, Per Hurtig and Anna Brunstrom
{toke.hoiland-jorgensen, per.hurtig, anna.brunstrom}@kau.se

Abstract

Several new Active Queue Management (AQM) and hybrid AQM/fairness queueing algorithms have been proposed recently. They seek to ensure low queueing delay and high network goodput without requiring parameter tuning of the algorithms themselves. However, extensive experimental evaluations of these algorithms are still lacking. This paper evaluates a selection of bottleneck queue management schemes in a test-bed representative of residential internet connections of both symmetrical and asymmetrical bandwidths as well as WiFi. Latency under load and the performance of VoIP and web traffic patterns are evaluated under steady state conditions. Furthermore, the impact of the algorithms on fairness between TCP flows with different RTTs, and also the transient behaviour of the algorithms at flow startup is examined. The results show that while the AQM algorithms can significantly improve steady state performance, they exacerbate TCP flow unfairness. In addition, the evaluated AQMs severely struggle to quickly control queueing latency at flow startup, which can lead to large latency spikes that hurt the perceived performance. The fairness queueing algorithms almost completely alleviate the algorithm performance problems, providing the best balance of low latency and high throughput in the tested scenarios. However, on WiFi the performance of all the tested algorithms is hampered by large amounts of queueing in lower layers of the network stack inducing significant latency outside of the algorithms' control.

1 Introduction

Ensuring low latency, and in particular *consistently* low latency, in modern computer networks has become increasingly important over the last several years. As more interactive applications are deployed over the general internet, this trend can be expected to continue. Several factors can contribute to unnecessary latency (for a survey of such factors, see [1]); in this paper we

focus on the important factor of excessive queueing delay, particularly when the network is congested.

Recent re-emergence of interest in the problem of congestion-induced excessive queueing latency has, to a large extent, been driven by the efforts of the bufferbloat community [2, 3], which has also worked to develop technical solutions to mitigate it. In short, bufferbloat is a term used to describe the effect that occurs when a network bottleneck is congested and large buffers fill up *and do not drain*, thus inducing a persistent queueing delay that can be much larger than the path round-trip time. Since the inception of the bufferbloat community effort, more and more people in both academia and industry are becoming aware of the problem; and several novel queue management schemes have been proposed to combat the problem.

These new queue management schemes seek to provide both low latency and high goodput, without requiring the extensive parameter tuning that was needed for earlier schemes like Random Early Detection (RED) [4]. The schemes include new Active Queue Management (AQM) algorithms, such as Controlled Delay (CoDel) [5] and Proportional Integral controller Enhanced (PIE) [6]. In addition, the older Adaptive RED (ARED) [7] algorithm has seen revival attempts for this use.

Most previous evaluations of these algorithms have been based on simulation studies. We extend this by comparing more algorithms (seven in total), both pure AQM algorithms and fairness queueing scheduling algorithms. In addition, we examine more traffic scenarios and application behaviours. Finally, we provide an updated examination of actual running code (the Linux kernel, version 3.14), which, due to the wide availability and open nature of the code, can be considered a real-world reference implementation for the algorithms. For all experiments, we provide access to the experimental data, and the tools to replicate them, online.[8]

We present our analysis in three separate parts: the Good, the Bad and the WiFi. First, the good: We compare steady state behaviour of the algorithms in a mix of traffic scenarios designed to be representative of a residential internet setting: measuring latency under load, and real-world application performance of VoIP and HTTP applications, with minimal tuning of the algorithms applied. The tested algorithms perform significantly better than FIFO queueing in these scenarios.

Second, the bad: We test the impact of the AQMs on fairness between TCP flows of unequal RTT, and analyse the transient behaviour of the algorithms when flows start up. We compare the goodput of four flows with RTTs varying almost two orders of magnitude. We find that the AQM algorithms exacerbate the tendency of unfairness between the TCP flows compared to FIFO queueing. We also look at the development of measured delay over time when competing TCP flows start up and start to claim bandwidth at the bottleneck link. This analysis shows that two of the AQM algorithms (PIE and CoDel) have severe issues in quickly controlling the induced delay,

[8] `https://www.cs.kau.se/tohojo/good-bad-wifi/`.

showing convergence times of several seconds with very high delay spikes when the flows start up.

Finally, the WiFi: Recognising that wireless networks play an increasing role in modern residential networks, we evaluate the algorithms in a setup where a WiFi link constitutes part of the tested path. We find that the algorithms fail to limit latency in this scenario, and it is quite clear that more work is needed to effectively control queueing in wireless networks.

The analysis of these three aspects of AQM behaviour contributes to a better understanding of residential network behaviour. It points to several areas that are in need of further evaluation and more attention from algorithm developers. One possible solution that has been deployed with promising results [8] is fairness queueing, exemplified by algorithms such as Stochastic Fairness Queueing (SFQ) [9] or the hybrid AQM/fairness queueing of fq_codel [10]. Hence, we have included three such algorithms in our evaluations along with the AQM algorithms. We find that they give vastly superior performance when compared with both FIFO queueing and the tested AQM algorithms, making the case that these types of algorithms can play an important role in the efforts to control queueing delay.

The rest of the paper is structured as follows: Section 2 discusses related work. Section 3 presents the experimental setup and the tested path characteristics, and Section 4 describes the tested algorithms. Section 5 presents the measurements of steady-state behaviour and their results, while Section 6 does the same for the experiments with fairness and transient behaviour. Section 7 covers WiFi and finally, Section 8 concludes the paper and outlines future work.

2 Related work

A large number of AQM algorithms have been proposed over the last two decades, employing a variety of approaches to decide when to drop packets; for a comprehensive survey, see [11]. Similarly, several variants of fairness queueing have been proposed, e.g. [12–14]. We have limited our attention to those algorithms proposed as possible remedies to the bufferbloat problem over the last several years. This section provides an overview of previous work on evaluating these algorithms and their effectiveness in combating bufferbloat.

The first evaluations of the AQM algorithms in question were performed by their inventors, who all publish extensive simulation results comparing their respective algorithms to earlier work [5–7]. All simulations performed by the algorithm inventors examine queueing delay and throughput tradeoffs in various straight-forward, mainly bulk, traffic scenarios. Due to being published at different times and with different simulation details, the results are not easily comparable, but overall, the authors all find that their proposed algorithms offer tangible improvements over the previously available algorithms.

In an extensive ns2-based simulation study of AQM performance in a cable modem setting [15], White compares CoDel, PIE and two hybrid AQM/fairness queueing algorithms, SFQ-CoDel and SFQ-PIE. Various traffic

scenarios were considered, including gaming, web and VoIP traffic as well as bulk file transfers. The simulations focus specifically on the DOCSIS cable modem hardware layer, and several of the algorithms are adjusted to better accommodate this. For instance, the PIE algorithm has more auto-tuning intervals added, and the fairness queueing algorithms have the number of queues decreased. The study finds that all three algorithms offer a marked improvement over FIFO queueing. The study concludes that PIE offers slightly better latency performance than CoDel but has some issues with bulk TCP traffic. Finally, the study finds that SFQ-CoDel and SFQ-PIE offers very good performance in many cases, but notes some issues in specific scenarios involving many BitTorrent flows.

Khademi et al [16] have performed an experimental evaluation of CoDel, PIE and ARED in a Linux testbed. The experiments focus on examining the algorithms at a range of parameter settings and measure bulk TCP transfers and the queueing delay experienced by the packets of the bulk TCP flows themselves. The paper concludes that ARED is comparable to PIE and CoDel in performance.

Rao et al [17] perform an analysis of the CoDel algorithm combined with a simulation study that compares it to the SFQ-CoDel algorithm. The paper concludes that SFQ-CoDel for many scenarios outperforms plain CoDel.

Järvinen and Kojo [18] perform a simulation study comparing PIE and CoDel to their own modified RED variant called HRED, focusing on transient load behaviour. They conclude that the CoDel algorithm does not scale with load, that PIE performs worse generally, but scales better, and that the HRED algorithm performs and scales better at transient loads.

Cai et al [19] employ fairness queueing to alleviate throughput unfairness between stations in a wireless network by applying it in a centrally controlled shaper. They find that this scheme can significantly reduce unfairness.

Finally, Park et al [20] perform a simulation study of CoDel on a wireless access point and concludes that, correctly configured, it can lower latency while keeping throughput high.

Our work expands on the above by (a) including more tested algorithms, also incorporating a variety of fairness queueing algorithms; by (b) testing a wider variety of traffic scenarios, in particular incorporating realistic application behaviour and looking at fairness issues and transient behaviour; and by (c) performing comprehensive, carefully designed tests of real-world implementations of the algorithms on actual networking hardware, while making the full data set and implementation available for scrutiny. We believe that together these factors make our evaluation an important contribution towards understanding the behaviour of modern queue management algorithms. In particular, we believe it is important to evaluate the algorithms in real-world implementations, to obtain a realistic view of their behaviour free from the idealisations imposed by purely simulation-based studies.

3 Experimental methodology

The experiments compare the selected queue management schemes in a variety of realistic scenarios mimicking a residential internet connection setting. This section presents the setup and methodology used to test the algorithms.

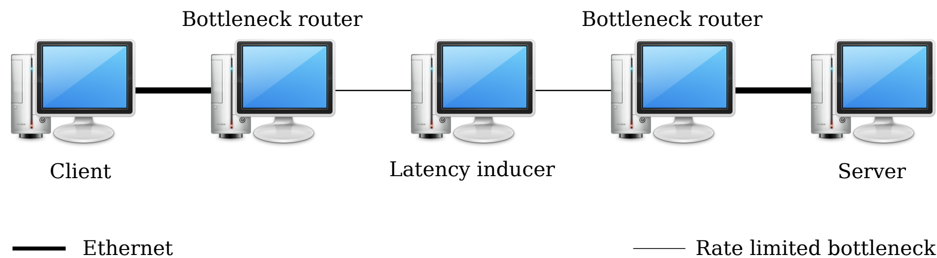


Figure 1: Physical test setup.

The tests are run in a controlled environment consisting of five regular desktop computers, as shown in Figure 1. The computers are equipped with Intel 82571EB ethernet controllers, and networked together in a daisy-chain configuration. This corresponds to a common dumbbell scenario, with the individual flows established between the endpoint nodes serving as multiple senders. The middle machine adds latency by employing the `dummynet` emulation framework [21]. The bottleneck routers employ software rate limiting (through the `tbf` rate limiter [22]) to achieve the desired bottleneck speeds. A separate control network is used to configure the test devices and orchestrate tests. All five computers run Debian Wheezy. The latency inducer runs the stock kernel (version 3.2) with the `dummynet` module added, while the others have had the kernel replaced with a vanilla kernel version 3.14.4. For the WiFi tests, a wireless link is added to the testbed (see Section 7).

The test setup is designed to correspond to a residential internet connection scenario. All tests are run with the bottleneck in three configurations: a symmetrical link at 100 Mbps, a symmetrical link at 10 Mbps, and an asymmetrical link with 10/1 Mbps download/upload speeds. The base RTT is set to 50 ms, corresponding to a mid-range internet latency. All TCP goodput values are measured at the application level; the bandwidth utilisation of the flows that measure latency is not counted.

The test computers are set up to avoid the most common testing pitfalls, as documented by the bufferbloat community in a best practices document [23]. This means that all hardware offload features are turned off, the kernel Byte Queue Limits have been set to a maximum of one packet and the kernel is compiled with the highest possible clock tick frequency (1000 Hz). All of these adjustments serve to eliminate sources of latency and queueing other than those induced by the algorithms themselves, for instance by preventing the network driver and hardware from queueing packets outside the control of the queue management algorithms. We have chosen this best-case configuration for our tests, because the object of interest is the behaviour of the algorithms

themselves, not the interactions between different layers of the operating system network stack and/or hardware. While turning off offloads and lowering the Byte Queue Limit settings can in some cases adversely affect achievable throughput, we have verified that our testbed has sufficient computational resources that this is not an issue at the speeds we are testing. All tests are run with both the CUBIC and New Reno TCP congestion control algorithms, but the results are only included here with the (for Linux) default CUBIC algorithm.

The tested queue management schemes are installed before the bottleneck link, in both the upstream and downstream directions. In a real residential setting this corresponds to service providers having the algorithms installed at their head end termination equipment, as well as in customer equipment. Many devices deployed in service provider networks do not run Linux, and so availability of an algorithm implementation in Linux does not necessarily translate directly to deployability today. However, since we are interested in assessing the *potential* benefits the algorithms can provide if deployed, we believe that testing in a scenario that grants the algorithms as much control of the bottleneck queues as possible is the right thing to do. We hope this can help make the case for implementing smarter queue management at the customer-facing side of operator networks. Until such implementations appear, Linux provides an intermediate queueing device that allows downstream shaping in the home gateway, which can help get queueing under control (with some limitations) [24].

The benchmarking tools used for the performance tests are the *Netperf* tool [25] for TCP traffic, the D-ITG tool [26] for generating VoIP streams and the cURL library for web tests [27]. The tests are run by means of a testing harness, *Flent* [28], which is available as open source software.

4 Tested algorithms

Seven queue management schemes, or *qdiscs* in Linux vocabulary, have been selected, including the default FIFO queueing mechanism. These represent algorithms that seek to function well with their default parameters at a wide variety of operating conditions in internet scale networks. While the parameter sensitivity of the algorithms is important, studies of this has been performed elsewhere (in e.g. [16]). Additionally, we believe performance at the default parameter setting is an important part of a queueing mechanism's overall performance (the difficulty of configuring RED has been cited as a major reason for its limited deployment [5]). For this reason, we focus on comparing the algorithm behaviours to each other with their default parameters. The drafts describing both the new AQMs (CoDel and PIE) include parameter settings known to work well in a wide variety of cases, and these values are also the defaults in the Linux implementation. We keep these defaults except where our test scenario is known to stray from the default operating range, or where no defaults exist.

All algorithms whose sole dropping mechanism is queue overflow (i.e., the pure packet schedulers), we have configured to have the same total queue length. This ensures that the scheduling behaviour is tested, rather than just the effects of different queue lengths. The lowest default value for these algorithms is used as the queue length, which is the SFQ default of 127 packets. This value is used at 1 and 10 Mbps; at 100 Mbps a longer queue size is required for TCP to fill the pipe. Thus, the queue size is increased to 1000 packets (the `pfifo_fast` default) at 100 Mbps.

Being available in mainline Linux, all the tested algorithms are available on a wide variety of platforms, and have been tested on a wide variety of hardware. In particular, they are part of the OpenWrt embedded router project, showing that running them on low-powered devices is quite feasible.

The algorithm parameters are summarised in Table 1 and the rest of this section describes each algorithm in turn.

Parameter		**1 Mbps**	**10 Mbps**	**100 Mbps**
pfifo_fast				
	`txqueuelen`	127	127	*1000*
ARED				
	`min`	1514	12500	125000
	`bandwidth`	1 Mbps	10 Mbps	100 Mbps
	`max`	3028	-	-
PIE				
	`target`	*20 ms*	*20 ms*	*20 ms*
	`tupdate`	*30 ms*	*30 ms*	*30 ms*
	`limit`	*1000*	*1000*	*1000*
CoDel				
	`target`	13 ms	*5 ms*	*5 ms*
	`interval`	*100 ms*	*100 ms*	*100 ms*
	`limit`	*1000*	*1000*	*1000*
SFQ				
	`limit`	*127*	*127*	1000
fq_codel				
	`target`	13 ms	*5 ms*	*5 ms*
	`interval`	*100 ms*	*100 ms*	*100 ms*
	`limit`	*10240*	*10240*	*10240*
fq_nocodel				
	`limit`	127	127	1000
	`interval`	100 s	100 s	100 s

Table 1: Qdisc parameters. Parameters that are kernel defaults are shown in *italics*. Some values are omitted here for brevity; see the published dataset and configuration scripts for details.

4.1 pfifo_fast

The pfifo_fast qdisc is the current default in Linux and consists of a three-tier priority queue with simple FIFO semantics. In the tests only one priority is used, so the qdisc can be viewed as a simple FIFO queue.

4.2 ARED

ARED is a dynamic configuration scheme for the RED AQM algorithm. It adjusts the RED max dropping probability based on the observed queue length, around a target point midway between the configured minimum and maximum queue sizes.

Following the configuration guidelines given in [7], the minimum queue size is set to half the target delay (queueing time being converted to a queue size by the link speed) and the max queue size is set to three times the minimum queue size. This makes the algorithm control point oscillate around the target delay size midway between the two values. A target delay of 20 ms is used, corresponding to the default for the PIE algorithm, which features a similar probabilistic drop scheme. However, at 1 Mbps, this would result in unachievable target queue size lengths of less than one Maximum Transmission Unit (MTU). To avoid this, at 1 Mbps the minimum and maximum queue size parameters are set to one and two MTUs respectively.

4.3 PIE

PIE is based on a traditional proportional integral controller design. It infers queueing delay from the instantaneous queue occupancy and the egress rate. The drop probability is then adjusted periodically (at a configurable interval defaulting to 30 ms) from the variations in the queueing delay over time, combined with a configured target delay, which defaults to 20 ms.

When PIE updates the drop probability, it does so based on the instantaneous estimated queueing delay and how it compares to the reference delay parameter and to the previously measured delay, respectively. Two parameters, α and β, control the weighing between the impact of these two differences on the calculated drop probability. PIE contains an auto-tuning feature which adjusts the values of α and β based on the measured level of congestion (expressed by the drop probability), setting the parameters higher when the network is more congested; this makes the algorithm react faster when the congestion level is higher. The Linux implementation has three levels of this auto-tuning, while more have been added in the version of PIE incorporated in the DOCSIS standard [29].

4.4 CoDel

CoDel seeks to minimise delay by directly measuring the time packets spend in the controlled queue. If this time exceeds a configured target for longer than a configured interval, packets are dropped at a rate computed by the interval

divided by the square root of the number of previous drops, until the queueing delay sinks below target again. The previous drop rate is then saved and the algorithm will start dropping again at the same level as before if it re-enters the drop state within a short time after having left it.

While the default values of 5 ms for target and 100 ms for interval are cited by the authors to work well for a large range of internet-scale bandwidths and RTTs, one known exception in the current implementation is when the minimum attainable queueing time (i.e., the transmission time of one packet) is higher than the target. In this instance, target should be set to the queueing time of one packet; thus, for the 1 Mbps tests, CoDel's target is raised to 13 ms.

4.5 SFQ

SFQ is a fairness queueing algorithm that employs a hashing mechanism to divide packets into sub-queues, which are then served in a round-robin manner. By default, packets are hashed on the 5-tuple defined by the source and destination IP addresses, the layer 4 port numbers (if available) and the IP protocol number, salted with a random value chosen at startup. The number of hash buckets (and thus the maximum number of active sub-queues) is configurable and defaults to 1024.

4.6 fq_codel

The fq_codel algorithm [10] is a hybrid algorithm consisting of a *flow queueing* scheduler which employs the CoDel AQM on each sub-queue. The *flow queueing* mechanism is a subtle optimisation of fairness queueing for sparse flows: A sub-queue will be temporarily prioritised when packets first arrive for it, and once it empties, a sub-queue will be cleared from the router state. This means that queues for which packets arrive at a sufficiently slow rate for the queue to drain completely between each new arrival, will perpetually stay in this state of prioritisation. The exact rate for this to happen depends on load, traffic and link characteristics, but in practice it means that many packets which impact overall interactivity (such as TCP connection negotiation and DNS lookups) get priority, leading to reduced overall application latency.

Additionally, fq_codel uses a *deficit* round-robin scheme when dequeuing packets from the sub-queues. This allows a queue with small packets to dequeue several packets each time a queue with big packets dequeues one, thus approximating byte-based fairness rather than packet-based fairness between queues. The granularity of the deficit mechanism can be set by a quantum parameter which defaults to one MTU.

4.7 fq_nocodel

The term 'fq_nocodel' is used to refer to the fq_codel algorithm configured so as to effectively disable the CoDel AQM (by setting the CoDel target parameter to be 100 seconds). This configuration is included to examine the

performance of the flow queueing mechanism of fq_codel, without having the CoDel algorithm operate on each queue. Since the queue overflow behaviour of fq_codel is very CPU-intensive,[9] this operating mode is not viable for deployment, but can be used in a controlled testbed environment with suitably over-provisioned CPU resources for the configured bandwidth.

5 The Good: steady-state behaviour

Steady-state behaviour is the most commonly assessed characteristic of queue management algorithms, and this is also the subject area of most analytical models (e.g. [30]). In this section we present three experiments examining the steady-state behaviour of the tested algorithms: one that looks at algorithm behaviour under synthetically generated load, and two that tests the impact of algorithms on performance of real-world application traffic. Each of the steady-state tests are run for 140 seconds (to minimise the impact of transient behaviour at flow start-up time) and repeated 30 times.

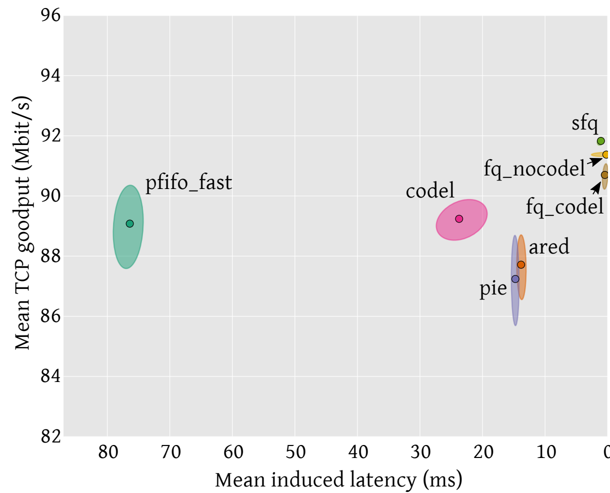
5.1 The Real-time Response Under Load test

The Real-Time Response Under Load (RRUL) test was developed by the bufferbloat community [31] specifically to stress-test networks and weed out undesirable behaviour. It consists of running four concurrent TCP flows in each direction, while simultaneously measuring latency using both UDP and ICMP packets. The goal is to saturate the connection fully, and the metrics of interest are TCP goodput, and the extra latency induced under load. The latter we define as the average observed latency under a full test run, minus the base path RTT. The RRUL test is also used as background traffic for the other steady-state tests below.

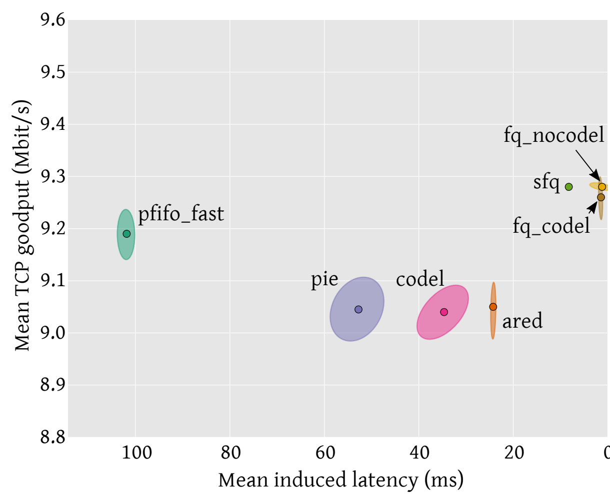
5.1.1 RRUL results

The results for the RRUL test are shown in Figure 2 as latency-goodput ellipsis graphs. The use of this type of graph was pioneered for visualising bandwidth/latency tradeoffs by Winstein in [32], and deliberately flips the latency axis to make better values be “up and to the right”. For the 10/1 Mbps link, in figure 2c, both upstream and downstream behaviours are shown on the same plot, reusing the same latency values for both. The results show that the default FIFO queue predictably gives a high induced latency, but with high goodput. An exception is on the asymmetrical 10/1 Mbps link, where the *downstream* goodput suffers slightly. This is due to ACKs being dropped in the upstream direction, preventing the downstream flows from fully utilising

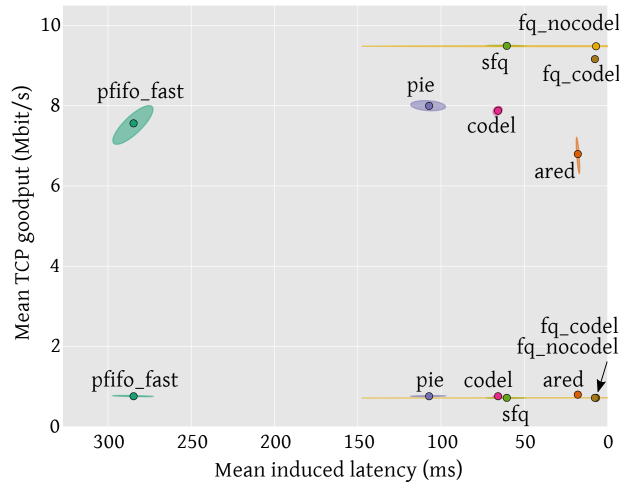
[9] When an overflow condition is detected, fq_codel linearly searches all available queues to find the longest one from which to drop a packet. This has a large impact, mainly by using up a lot of CPU cache. The implementors found this to have acceptable performance as long as it is used as a fallback mechanism to avoid overflow rather than as the main drop mechanism. Thus, changing the implementation to a more efficient drop mechanism would be advisable for a deployment scenario.



(a) 100/100 Mbps link speed, one direction shown.



(b) 10/10 Mbps link speed, one direction shown.



(c) 10/1 Mbps link speed, both directions shown.

Figure 2: (a and b) The RRUL test results, showing the median values and 1-σ ellipses of the per-test-run mean goodput and mean induced latency. (c) As (a and b), but showing both upstream and downstream traffic, re-using the same latency values.

the available bandwidth, and the behaviour is consistent with previous studies of TCP on asymmetric links [33]. The same effect is apparent for the ARED AQM, which achieves an even lower goodput, but at the same time it achieves a lower latency.

All the AQMs achieve lower queueing delay than FIFO queueing, and the newer AQMs fare better goodput-wise at the low bandwidth. The difference between the steady-state behaviours of the three AQMs can be explained as follows: ARED and PIE are both designed to control the average queue length around a set point. ARED controls the drop probability based on how the average queue length deviates from the desired set-point, scaling the drop probability rapidly as the queue fluctuates in a rather narrow interval around the target. This causes it to be fairly aggressive, achieving low delays, but at a cost in throughput. This is particularly apparent at 1 Mbps, where the size of the interval is a single packet.

PIE, on the other hand, adjusts its drop probability based on both the queue's deviation from the set-point and the previous delay values, and the drop probability is adjusted less often. Together, this leads to a smoother oscillation around the target, and a less aggressive behaviour. At 100 Mbps, however, PIE shows a more aggressive drop behaviour than at lower bandwidths. This is most likely due to the fact that the built-in auto-tuning of PIE (which scales the drop probability adjustment parameters α and β with the observed drop probability) is too narrow in scope. The auto-tuning consists of a lookup table for drop probabilities in ranges starting from 0.1%, with lower drop probabilities resulting in a slower adjustment. However, everything below 0.1% is treated the same, and since the steady-state drop probability of a 100 Mbps link is markedly lower than 0.1%, this results in the algorithm reacting more aggressively than it does at lower bandwidths.

Finally, CoDel uses its *target* parameter as a lower bound on how much latency to tolerate before reacting by dropping packets. This means that the set-point does not function as an average around which to control the queues, as the other algorithms do. Instead, the queue is controlled to an average somewhat above the target. The auto-tuning of the interval from the drop count then serves to find the right drop rate, and CoDel oscillates in and out of drop mode in the steady state. This leads to a steady-state performance midway between ARED and PIE (excluding the 100 Mbps PIE behaviour), as seen from the figure.

The highest goodput of all the configured queue management schemes, however, is achieved by the AQM-less fairness queueing algorithms, with fq_codel lagging a tiny bit behind. This indicates that with the flow isolation offered by fairness queueing, additional drop signals from an AQM hurt throughput with no gain in terms of lower queueing delay for competing flows. However, this is offset by the fact that fq_codel keeps the TCP window significantly smaller than fq_nocodel, meaning that the TCP flows themselves experience less queueing latency. This can be important for interactive applications that also transfer enough data to induce queueing, such as screen sharing applications or adaptive rate video streaming.

At 100 Mbps link speed, all three fairness queueing algorithms show comparable (and very close to zero) induced latency. However, at the lower bandwidths where the time to transmit single packets can be noticeable, it is clearly seen how it is beneficial that the flow queueing mechanism prioritises the sparse flows measuring latency, resulting in practically zero induced latency. The high variance of the fq_nocodel algorithm at the lowest speed results from a hash collision between a latency measurement flow and a data flow, resulting in one of the test runs exhibiting high latency.

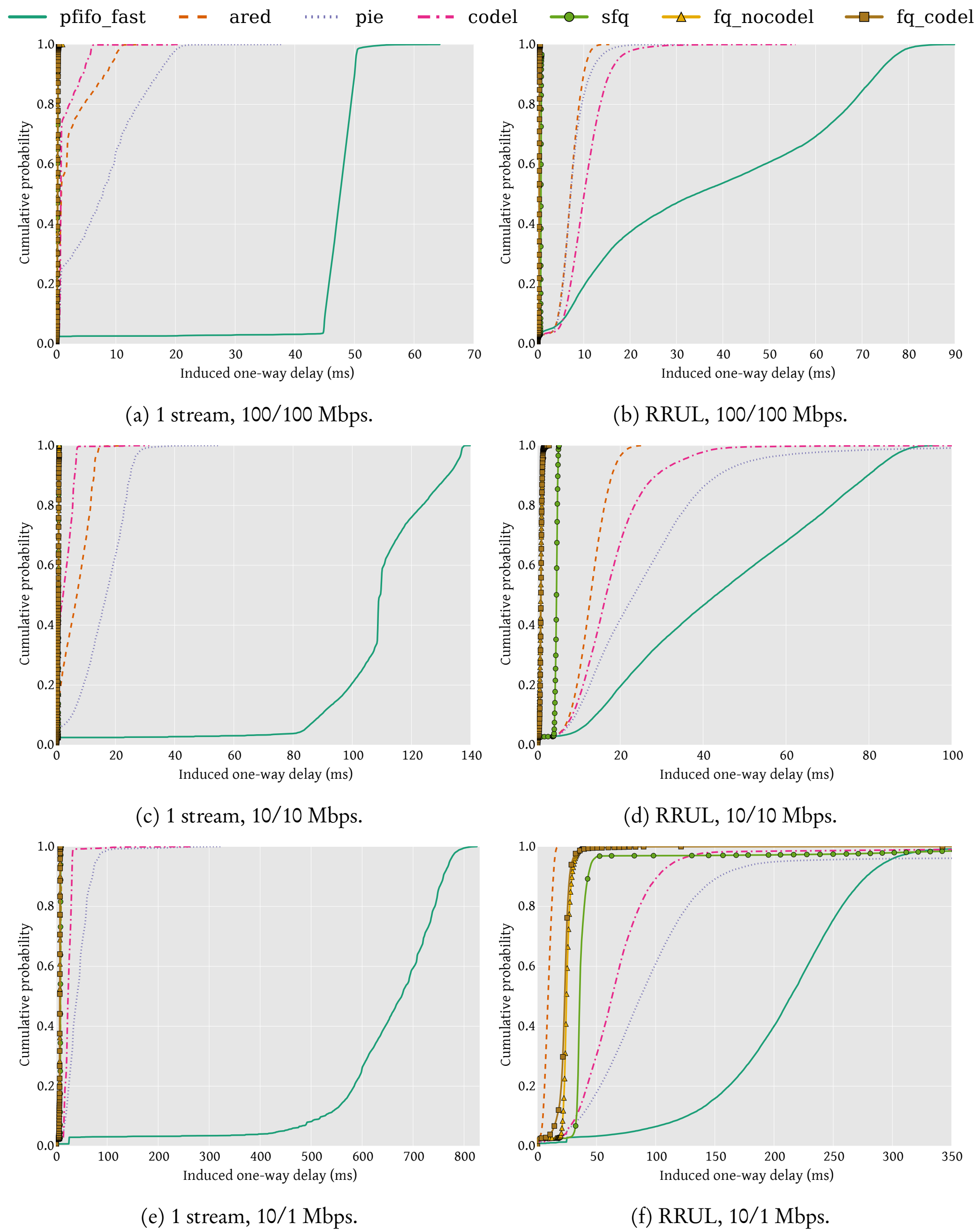
5.2 VoIP test

The VoIP test seeks to assess the performance of voice traffic running over a bottleneck managed by each of the queue management schemes. This is done by generating synthetic VoIP-like traffic (an isochronous UDP flow at 64 Kbps) in the upstream direction, and measuring the end-to-end one-way delay and packet loss rate. The test is performed with one competing TCP flow in the same direction as the VoIP flow, as well as with the full RRUL test as background traffic on the link.

5.2.1 VoIP results

The results for the VoIP tests are shown in Figure 3. The graphs show the CDF of the one-way delay samples of the VoIP traffic with a 200 ms sampling interval. The accompanying TCP goodput results are omitted for brevity, but the relative goodput for each algorithm mirror those from the RRUL test discussed above. For latency, the results mirror those of the RRUL tests: new AQM algorithms give a marked improvement over FIFO queueing, but with their respective latency values varying depending on the link bandwidth and cross traffic. And as before, the fairness queueing give the best latency results. However, it is interesting to note that the effects of hash collisions in the queue assignments are apparent in the RRUL results at 1 Mbps, heavily influencing the performance of SFQ and fq_nocodel. CoDel and PIE also show a long tail of delay values at 1 and 10 Mbps for RRUL, corresponding to the transient delay (see Section 6.2).

Loss statistics are shown in Table 2. From these, it is quite apparent that the AQMs would render a VoIP conversation completely hopeless at 1 Mbps, even with only a single competing flow. With RRUL as cross traffic it is even worse, with the FIFO queueing also showing high loss rates. Additionally, ARED shows loss in excess of 25%, explaining its very low delay values. For all tests, the flow isolation of the fairness queueing algorithms effectively protect the VoIP flows from loss, with the exception of SFQ and fq_codel at 1 Mbps with RRUL as cross-traffic. This can be explained by the fact that at this speed, the time to transmit a packet is a significant component of the latency, adding enough delay for the VoIP flow to build a bit of queue and hence suffer loss.



(a) 1 stream, 100/100 Mbps.

(b) RRUL, 100/100 Mbps.

(c) 1 stream, 10/10 Mbps.

(d) RRUL, 10/10 Mbps.

(e) 1 stream, 10/1 Mbps.

(f) RRUL, 10/1 Mbps.

Figure 3: VoIP test results. The CDF plots show the distribution of induced one-way delay over all samples from the VoIP streams.

Table 2: VoIP average packet loss over all test runs. A '–' indicates no packet loss.

	1 stream cross traffic		
	1 Mbps (%)	10 Mbps (%)	100 Mbps (%)
pfifo_fast	0.88	0.10	–
ARED	7.95	0.45	0.002
PIE	2.75	0.04	0.002
CoDel	6.46	0.02	0.002
SFQ	–	–	–
fq_-nocodel	–	–	–
fq_codel	–	–	–
	RRUL cross traffic		
	1 Mbps (%)	10 Mbps (%)	100 Mbps (%)
pfifo_fast	8.54	0.20	0.032
ARED	26.33	0.61	0.019
PIE	14.03	0.44	0.016
CoDel	10.60	0.19	0.004
SFQ	0.42	–	–
fq_-nocodel	–	–	–
fq_codel	0.04	–	–

5.3 Web test

The web test measures the web browsing performance of a user accessing the web through a bottleneck equipped with the tested queue management schemes.

To retrieve a web site, web browsers commonly first lookup the site host name, then retrieve the main HTML document, and finally retrieve all the resources associated with the document over several concurrent connections. Since web browsers continue to evolve at a rapid pace, and so constitute somewhat of a moving target, we have chosen to focus on this network-centric behaviour as a way to approximate real web behaviour. We simply define the *page fetch time* as the total time to retrieve all objects of each web site. This metric also has the added benefit of being reproducible without relying on a specific implementation of a particular browser or rendering engine. We have chosen the well-tested and widely used cURL library [27] as the basis for our test client [34], which mimics this fetching behaviour in a reproducible way (a feature we were not able to find in any existing web benchmarking tools).

Two web pages of different sizes are mirrored on the test server: the Google front page (56KB data in a total of three requests) and the front page of the Huffington Post web site (3 MB in a total of 110 requests).[10] We believe these

[10]The test pages are henceforth referred to as 'Google' and 'Huffpost', respectively.

two sites are well-suited to represent opposite ends of the web scale: a small interactive page and a large and complex site with many elements to be fetched.

The tested web site is repeatedly fetched throughout the duration of the test run. The metric of interest is the page fetch time mentioned above. The test is run both with the RRUL test as background traffic, and with a single TCP flow in the *upstream* direction, competing with the HTTP *requests* going to the web server. The latter is included to show the importance of having timely delivery of the HTTP requests, and how failure to achieve this can negatively impact the entire web browsing performance.

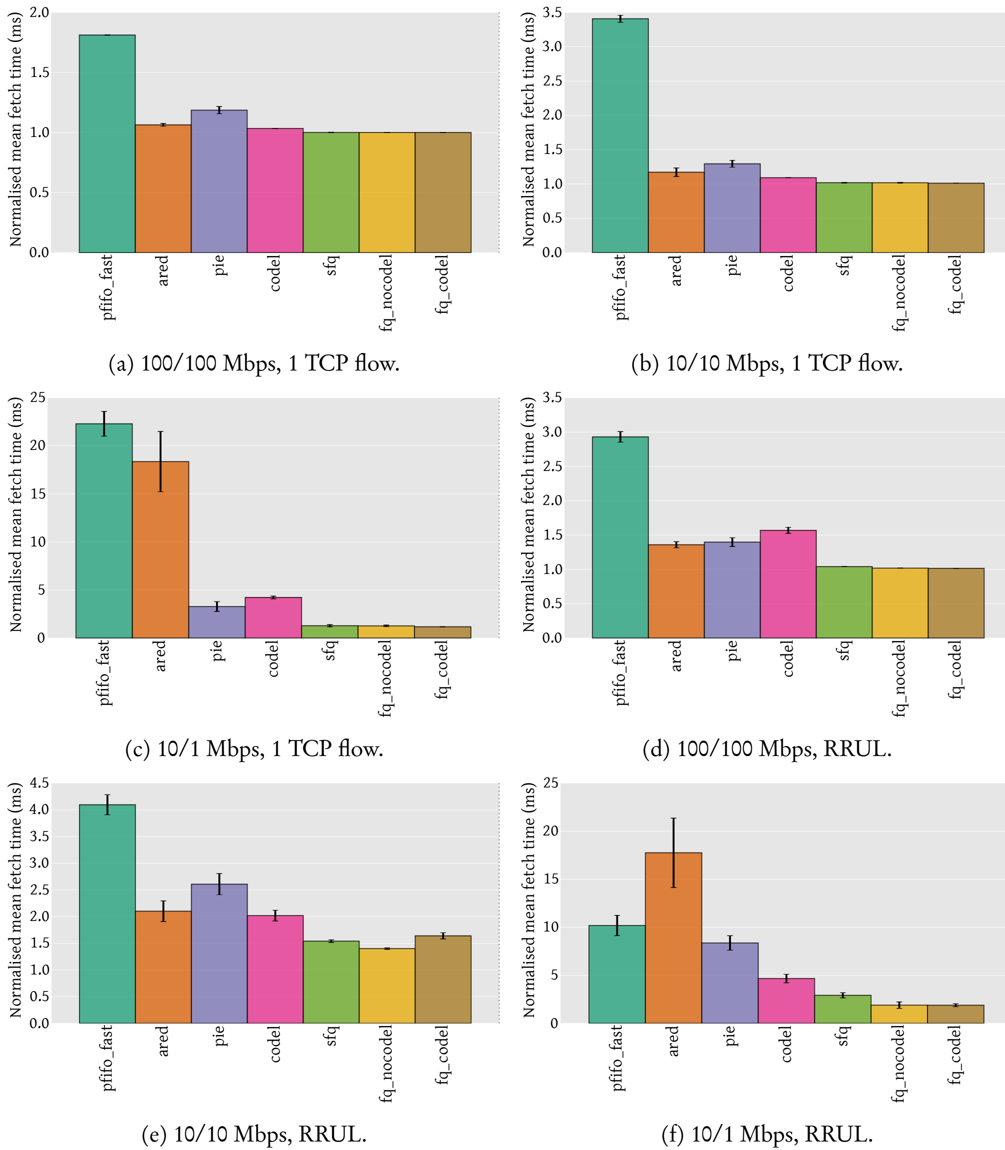
5.3.1 Web results

The results for the web tests are shown in Figures 4 and 5. For each test run, the average fetch time is computed, and the mean and standard deviation of these averages over the test repetitions are displayed on the result graphs.

The results show that managing delay greatly impacts web browsing performance in a positive way. However, one exception is the ARED algorithm at low bandwidths: here, performance is both highly variable and sometimes even worse than the FIFO queue. This is caused by a too aggressive drop behaviour, which causes SYN packets in the HTTP requests to be lost, requiring retransmission. This effect is most pronounced on the simpler Google page, where the total fetch time is more affected by timely delivery of the HTTP request.

SYN losses are also the reason that the FIFO queue shows worse behaviour with a single TCP flow as cross traffic than with the full RRUL test. We attribute this to the fact that with the RRUL test, a lot of the queue space in the upstream direction is occupied by small ACK packets, which take less time to put on the wire. When the queue is full and a full-sized packet is at the front of the queue, it stays full for the entire time it takes to dequeue that one packet. This means that the smaller the packets, the shorter the average time before a new queue space opens up, and hence the better the chance that the SYN packet gets a space in the queue upon arrival.

Another interesting feature of the result is that *any* queue management significantly improves this important real-world application performance. The performance differences between the AQM algorithms and the fairness queueing schemes are in many cases less pronounced than in the other tests, since all the algorithms achieve sufficient latency reduction to get the fetch time very close to the unloaded case. For those cases where the fetch time is significantly higher than the unloaded case, the performance differences are more pronounced. The odd case out is Huffpost at 10 Mbps with the RRUL test, where the fairness queueing algorithms show worse performance than CoDel and PIE. This is most likely because the Huffpost site consists of many objects that need to be fetched: They are each fairly small and so will be sent in a single burst of packets. The bursts go into the single queues back-to-back, whereas per-flow fairness imposed by the fairness queueing algorithms split them up causing a longer total completion time.



(a) 100/100 Mbps, 1 TCP flow.

(b) 10/10 Mbps, 1 TCP flow.

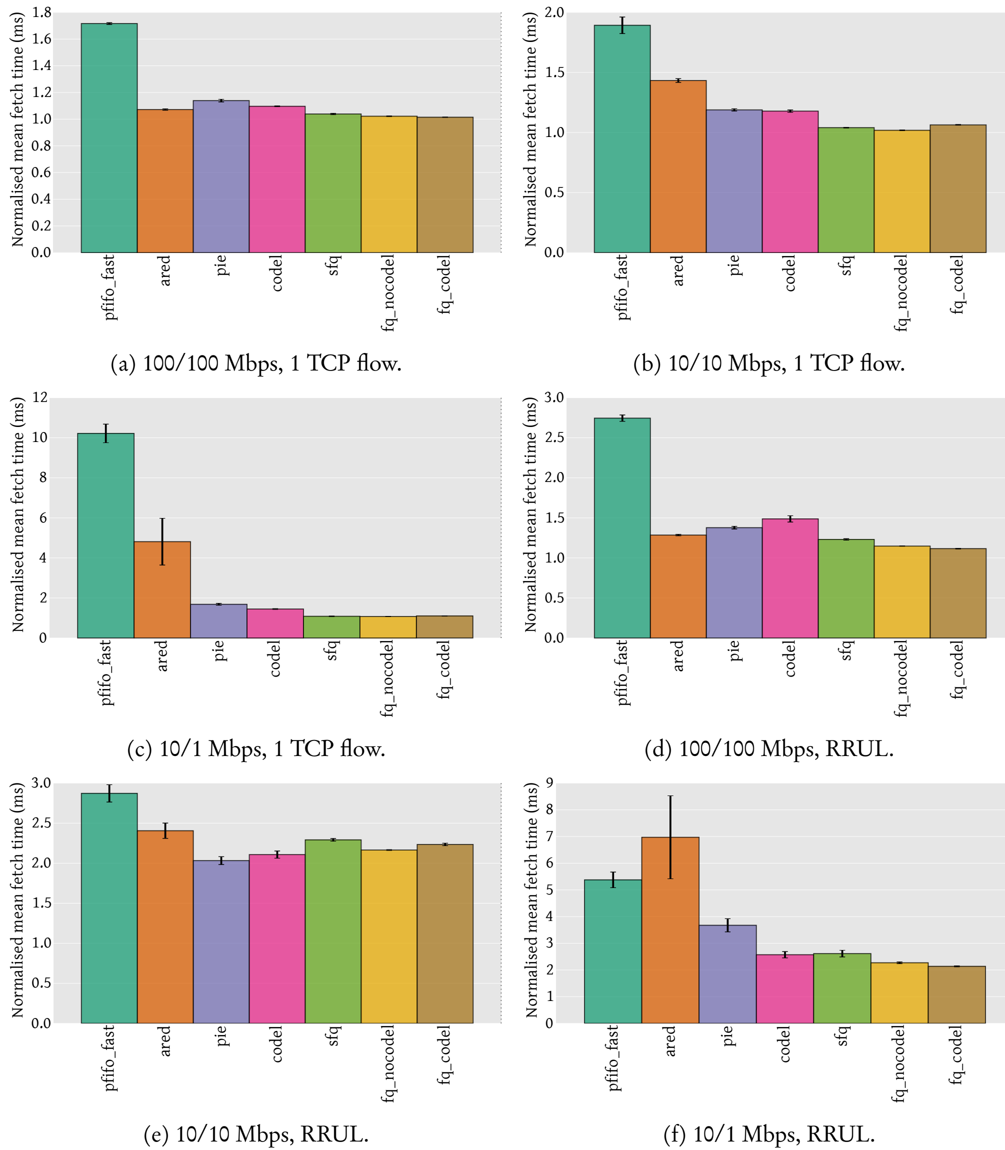
(c) 10/1 Mbps, 1 TCP flow.

(d) 100/100 Mbps, RRUL.

(e) 10/10 Mbps, RRUL.

(f) 10/1 Mbps, RRUL.

Figure 4: HTTP mean fetch times for Google. The upper row shows results for the tests with a single TCP flow as cross traffic, while the lower row shows results for tests with the RRUL test as cross traffic.



(a) 100/100 Mbps, 1 TCP flow.

(b) 10/10 Mbps, 1 TCP flow.

(c) 10/1 Mbps, 1 TCP flow.

(d) 100/100 Mbps, RRUL.

(e) 10/10 Mbps, RRUL.

(f) 10/1 Mbps, RRUL.

Figure 5: HTTP mean fetch times for Huffpost. The upper row shows results for the tests with a single TCP flow as cross traffic, while the lower row shows results for tests with the RRUL test as cross traffic.

5.4 Discussion

The steady state test results show that a marked improvement is possible by managing the bottleneck queues. All three AQM algorithms show consistent improvements over FIFO queueing, although the older ARED algorithm exhibits a tendency to drop too aggressively, as does PIE at 100 Mbps.

Together, the steady state results underscore the benefit of deploying AQM in place of the prevalent FIFO queues of today's networks; this is in broad agreement with previous studies. It is worth noting, however, that ARED does require quite a bit of parameter tuning compared to the two other algorithms. In particular, parameters need to be set corresponding to the link bandwidth, which makes the algorithm somewhat more complex to deploy than the others.

The analysis of the fairness queueing algorithms show very impressive performance. At no point are the fairness queueing algorithms out-performed by the AQM algorithms, and in most cases fairness queueing outperforms AQM by a large margin. For VoIP traffic in particular, the flow isolation prevents the VoIP flows from experiencing a loss rate that, at the lowest bandwidth, would make any conversation completely untenable. This indicates that various forms of fairness queueing have an important role to play in dealing with queueing-induced latency. The sparse flow optimisation of the fq_codel flow queueing algorithm provides a marked additional improvement on top of regular fairness queueing, especially at lower bandwidths.

6 The Bad: fairness and transient behaviour

Two aspects of queue management are often overlooked when evaluating queue management algorithms: the algorithms' influence on inter-flow fairness, and the transient behaviour exhibited when flows start up. In this section we present our analysis of these two aspects of the behaviour of the tested algorithms.

6.1 Inter-flow fairness

It is well-known that fairness queueing algorithms can improve flow fairness characteristics [35], and indeed it is a design goal for such algorithms (hence the term *fairness queueing*). However, fairness characteristics of pure AQM algorithms are not well understood. In this section, we investigate fairness behaviour of all the tested algorithms.

We do this by means of the RTT-fairness test, which examines the RTT fairness properties of TCP under each of the queueing algorithms. It is well-known that the TCP goodput is affected by the RTT [36], because the congestion control algorithm reacts to feedback that is on an order of the RTT. While TCP CUBIC is designed to improve RTT fairness [37], some RTT fairness issues still remain [38]. The purpose of the RTT-fairness test is to evaluate whether the queue management schemes make this effect worse, or whether they help

alleviate it. The test consists of running four concurrent TCP streams from the client to the server, each with a different RTT value (10, 50, 200 and 500 ms respectively), and measuring the aggregate TCP goodput of each stream. To minimise the impact of transient effects from the initial TCP ramp-up even at the long base RTT, the test length is increased to 600 seconds for this test. As expected, the RTT fairness characteristics of the CUBIC and New Reno congestion controls differ. However, this is only a difference in magnitude, and does not influence the relative performance of the algorithms compared to each other. We have thus omitted the Reno results for brevity.

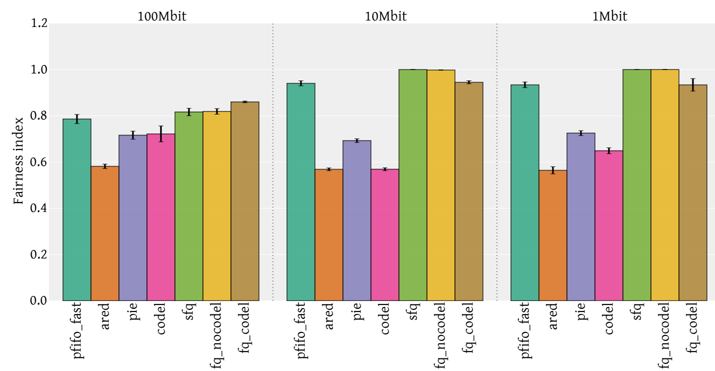
6.1.1 RTT Fairness results

Figure 6 shows the test results for the RTT fairness tests. The figure shows Jain's fairness index [39] calculated over the goodput values of the four competing flows, as well as the total goodput of each of the four flows. For each test repetition, the total goodput value for each flow is used; all graphs show the mean and standard deviation over the test repetitions.

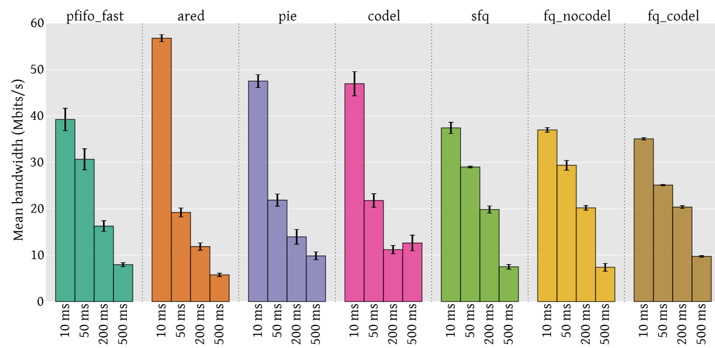
The AQM algorithms exhibit a tendency to *worsen* the RTT-unfairness of TCP, compared to the FIFO queue. This can be clearly seen by comparing the throughput of the flows with the highest latency between the algorithms. This is due to several factors: Firstly, the added queueing latency of the FIFO queue serves to even out the RTT differences of the different flows. Furthermore, packet traces reveal that the AQM algorithms cause the long-RTT flows to experience loss at an even rate throughout the test, whereas FIFO queueing results in bursty losses, from which TCP recovers better. Finally, the AQMs tune themselves to the shorter flow RTTs to control the queue, hurting the flows with longer RTT which share the queue. Together, these effects combine to lower the fairness rating of the AQM algorithms.

As expected, and in contrast to the AQM results, the fairness queueing algorithms achieve very good fairness results. The pure schedulers with no AQM achieve perfect fairness, which is to be expected from their round-robin scheduling behaviour. The fair results of fq_codel is worse than for the other scheduling algorithms, for the same reason as stated above: CoDel fails to tune itself to the very short and very long RTTs in this test. This results in the bandwidth distribution of the flows getting skewed, leading to worse fairness results. At 100 Mbps, the schedulers fail to exhibit perfect fairness behaviour, because at this bandwidth their total queue space is too small for the flows with long RTTs to effectively use the available bandwidth.

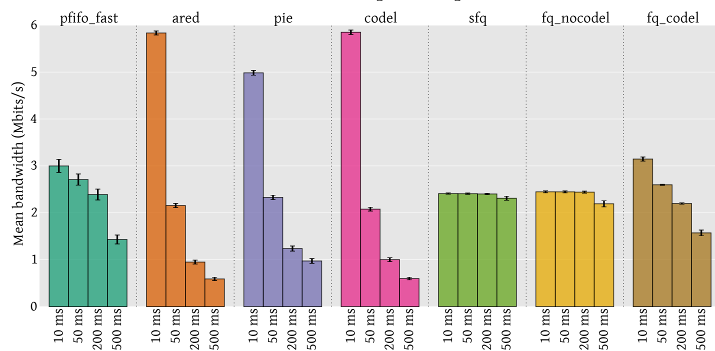
One peculiar feature of the results is that at 1 Mbps, FIFO queueing, ARED and fq_codel all show lower aggregate throughput for the 10 ms RTT flow than for the flow with a 50 ms RTT. This has different explanations for each of the algorithms. For FIFO queueing, this happens because the short-RTT flow initially ramps up its congestion window, then suffers a series of consecutive congestion events which causes it to lower its window to a level it never recovers from. For ARED, the high drop rate causes the low-RTT flow to suffer a series of consecutive retransmission timeouts, causing throughput to drop. For fq_codel, the short flow tends to suffer retransmission timeouts,



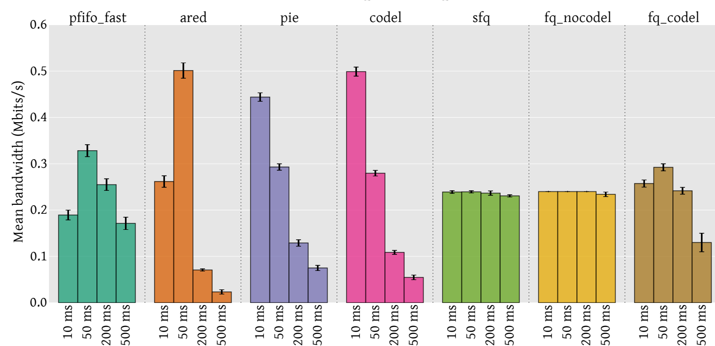
(a) Fairness index.



(b) 100/100 Mbps link speed.



(c) 10/10 Mbps link speed.



(d) 10/1 Mbps link speed.

Figure 6: The RTT fairness test results. (a) Jain's fairness index as computed from the goodput values of each flow. (b–d) The mean goodput of each of the four TCP streams for each bandwidth.

because its BDP is so small (312 bytes) that it rarely has enough outstanding data to trigger fast retransmit when a packet is dropped by CoDel in the middle of a window, but because it has to wait its turn in the round-robin scheduler with the other flows, each packet experiences enough queueing latency to trigger the drops. For both ARED and fq_codel, this also causes a drop in total throughput, with ARED losing just over 10%, while fq_codel loses around 5%. All other algorithms have identical total throughput for each bandwidth.

6.2 Transient behaviour

The transient behaviour of queue management algorithms is often overlooked in evaluations that all too often focus mainly or exclusively on steady state behaviour. Analytical models of transient behaviour are almost entirely non-existent, but also simulation-based and experimental evaluations often overlook this. However, transient behaviour can be vital for the overall perceived performance of the network: an algorithm that keeps latency low in the steady state but fails every time a transient event occurs makes for a quite bad overall user experience. In this section we investigate an extreme case of transient behaviour: what happens to the measured delay when the four bi-directional TCP streams of the RRUL test start up.

6.2.1 Transient behaviour results

Figure 7 shows the results of the transient behaviour tests. This shows simply a time sequence graph of the measured latency over the first 25 seconds of an RRUL test run. The values are point-wise averages over the 30 iterations.

The results show that both CoDel and PIE have severe problems keeping the delay low when the TCP flows start up. At the lower bandwidths, PIE has the worst behaviour, with delay sky-rocketing and even temporarily being higher than for the FIFO queue in the 10 Mbps tests. CoDel fares somewhat better relative to PIE at the lower bandwidths, but significantly worse at 100 Mbps. They both take from several seconds up to more than 20 seconds to get latency back under control, which is a significant impact on the user experience and can easily lead to an almost perpetual state of high delays.

These delay spikes in the traffic managed by CoDel and PIE have a common cause: The four simultaneous flows in slow start are simply overwhelming the algorithm control mechanisms, which do not tune the drop rate quickly enough to the new environment. For both algorithms, part of the reason is that the algorithms do not engage at all within the first 100 ms (PIE has a burst allowance of 100 ms, and CoDel's interval is 100 ms), at which point the queue is already substantial.

Additionally, for CoDel it is noticeable that the time it takes to get the delay under control goes *up* with the link bandwidth. This corresponds to the fact that the rate at which CoDel increases its drop rate is linear, and proportional to the inverse of the link speed [40]. So in other words, the initial spikes in latency seen by the CoDel-controlled flows occur because

CoDel's drop rate is increased too slowly, and at a rate that is dependent on link bandwidth.

Similarly, for PIE, the drop probability increase is capped to two percentage points in each update cycle, in order to protect single TCP flows in slow start from experiencing timeouts [41]. In our case of four simultaneous flows starting up, this results in a marked delay in getting latency under control. Interestingly, PIE contains another optimisation that will increase the drop probability rapidly when the absolute delay exceeds 250 ms, which corresponds to the size of the delay spike we see at 10 Mbps. At 100 Mbps, the relative lack of a delay spike for PIE corresponds to the more aggressive behaviour PIE exhibits at this bandwidth, as noted earlier.

The ARED algorithm fares significantly better and shows almost no delay spike but instead jumps smoothly to the steady state delay values. The fairness queueing algorithms simply assign the newly started flows their own queues, and so they do not impact the latency measurements at all, even in the slow start phase.

6.3 Discussion

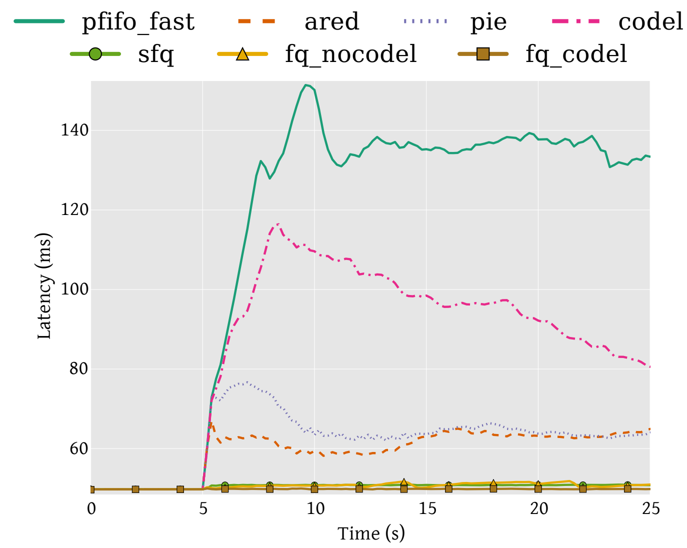
The fairness results are an example of a metric where the AQM algorithms actually exhibit worse behaviour than FIFO queueing. The fairness aspect is often overlooked in evaluations of AQM algorithms, but can be an important factor especially when considering deploying an AQM algorithm on a link likely to see traffic with highly varying RTT.

Likewise, the transient results reveal a potentially quite severe limitation of the new AQM algorithms, which can take several seconds to get delay back under control after a significant change in conditions occurs. An obvious real-world example of such behaviour is web browsing, where a browser initiating a large page download over several simultaneous connections easily can result in behaviour similar to that seen here.

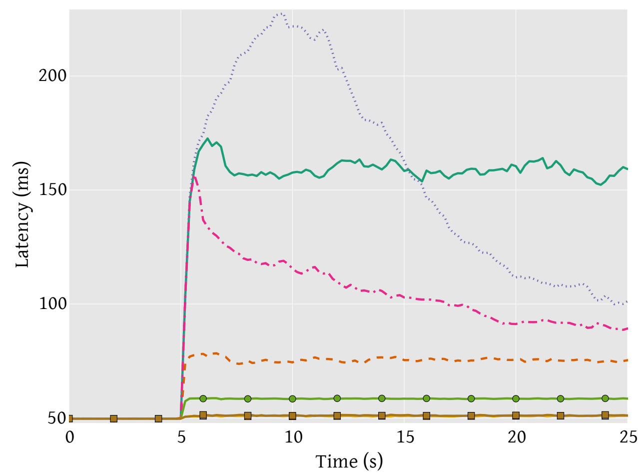
Together, these two aspects highlight areas that need more attention in future AQM research. Additionally, both are areas where the flow isolation provided by fairness queueing algorithms proves to be a very effective remedy. This makes the case for having such algorithms play an important role in managing queueing delay.

7 The WiFi: adding a wireless link

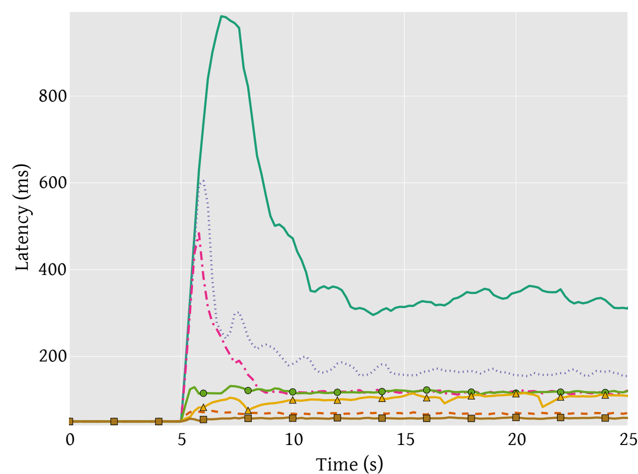
An increasing share of traffic in the home goes via wireless connections. This can influence the behaviour of queue management algorithms by moving the bottleneck to the WiFi link. If this happens, then even if the queue management algorithms are applied to the WiFi link, their behaviour can differ because the characteristics of the physical link is different (most notably, WiFi protocols include retransmit and packet aggregation features which can both affect latency and queueing). To test this scenario, we have added a WiFi



(a) 100/100 Mbps link speed.



(b) 10/10 Mbps link speed.



(c) 10/1 Mbps link speed.

Figure 7: The transient behaviour of the algorithms. The plots show the delay development over time for the first 25 seconds of the RRUL test. Each line is the (point-wise) mean of the test runs for each algorithm.

link to the testbed, and run the same sets of tests in this modified scenario. The modified test setup is shown in Figure 8.

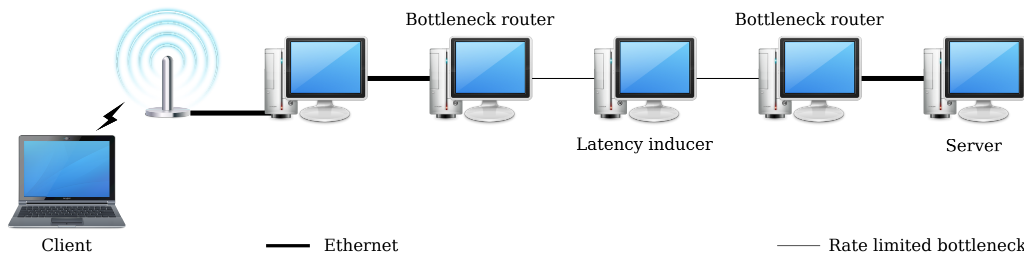


Figure 8: WiFi test setup

We use an Ubiquiti Nanostation M5 access point running OpenWrt 14.07 and using the *ath9k* WiFi driver. The client is a laptop running the same Debian version and kernel as the rest of the testbed. The laptop is equipped with an Intel WiFi Link 5100 card using the *iwlwifi* driver. The test is performed using 802.11n on an empty channel in the 5 GHz frequency spectrum. Rather than place the laptop and access point right next to each other, we have placed them on opposite sides of a wall. We believe this setup approximates a residential usage scenario reasonably well, with the exception that the clear channel is likely to lead to better results than in, say, a crowded apartment building with dozens of WiFi networks. We apply the queue management algorithms to both sides of the WiFi link as well as to the bottleneck link as before.

On this WiFi setup we have re-run all tests designed to test a single link characteristic, i.e. everything except the fairness test. However, for the lower bandwidths, the WiFi link does not constitute a bottleneck, and so we see no meaningful difference in the results.[11] For this reason, we have omitted those results and only include the results for the 100 Mbps bottleneck link. Furthermore, as can be seen in the following, the RRUL test results show such high induced latency that the transient spikes seen in the previous section are absent for the WiFi results. This, too, has thus been omitted.

In the following, we present the results of the WiFi evaluation, in the same order as the previous sections.

7.1 The RRUL test

The RRUL results are shown in Figure 9. A couple of interesting features are clearly visible on this graph. Firstly, the algorithms show the same ordering of latency behaviour, with FIFO being worst, followed by PIE and CoDel, the ARED and the fairness queueing algorithms. However, the magnitude of induced latency is different, with the lower bound being around 100 ms. We attribute this to queueing in lower layers (i.e. in the driver and hardware)

[11] Looking at the detailed behaviour over time, we see a small number of delay spikes for the low-bandwidth tests, which we attribute to WiFi retransmissions. However, these spikes are so few in number (and so small that they only show up on the fairness queueing results) that they do not impact the aggregate behaviour of the algorithms.

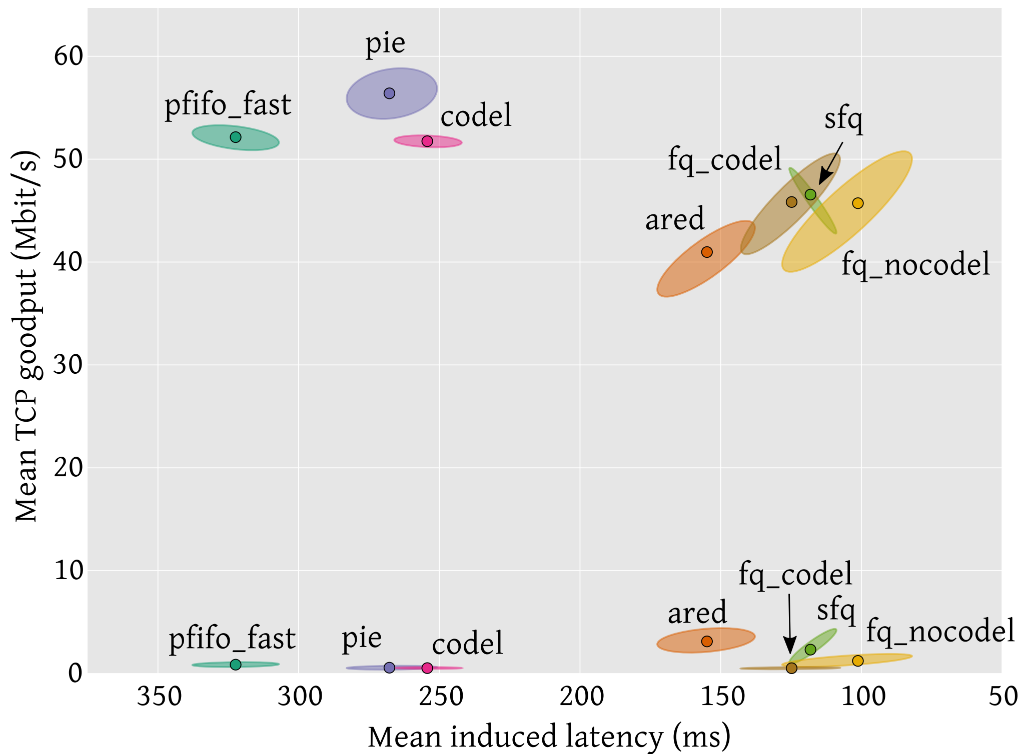


Figure 9: RRUL results for the WiFi setup. The top part is downstream traffic, the bottom part upstream.

which the queue management algorithms cannot control. Linux's Byte Queue Limits [42] mechanism is designed to deal with this in Ethernet drivers, however no such mechanism exists for WiFi, and it is doubtful whether the same mechanism can be applied, due to the aforementioned packet aggregation and retransmit features.

The second noteworthy feature of the RRUL results is that upstream throughput drops to almost nothing, even though the link nominally has the same bandwidth in both directions. This is a consequence of air-time unfairness, and for this particular combination of devices and drivers, it is hurting the upstream direction. Testing of other devices in the bufferbloat community has shown that this can just as well be seen in the other direction.

7.2 VoIP traffic

The VoIP WiFi results are shown in Figure 10. They show that when there is only a single flow as competing traffic, the queue management schemes exhibit almost completely identical behaviour, confirming the view that the induced delay is in layers below the qdisc layer where the algorithms cannot control it. When the RRUL test is used as cross traffic, the delay results match those from the RRUL test itself. The loss results (in Table 3) show a small loss ranging between 0.2% and 0.5% for one stream, and very high loss percentages for

the AQMs with the RRUL test, corresponding to the low effective upstream bandwidth.

Table 3: VoIP average packet loss over all WiFi test runs

	VoIP packet loss	
	1 stream (%)	RRUL (%)
pfifo_fast	0.34	16.66
ARED	0.13	5.30
PIE	0.18	27.52
CoDel	0.19	18.56
SFQ	0.47	1.71
fq_nocodel	0.17	1.59
fq_codel	0.22	2.64

7.3 Web results

The web results from the WiFi tests are shown in Figure 11. These show that once again, for one upload stream, the result is determined by something other than the active queue management algorithm. The relative positions of the different algorithms with the RRUL test as cross traffic match those for the wired tests at 100 Mbps, except that PIE and CoDel's disadvantage is more pronounced.

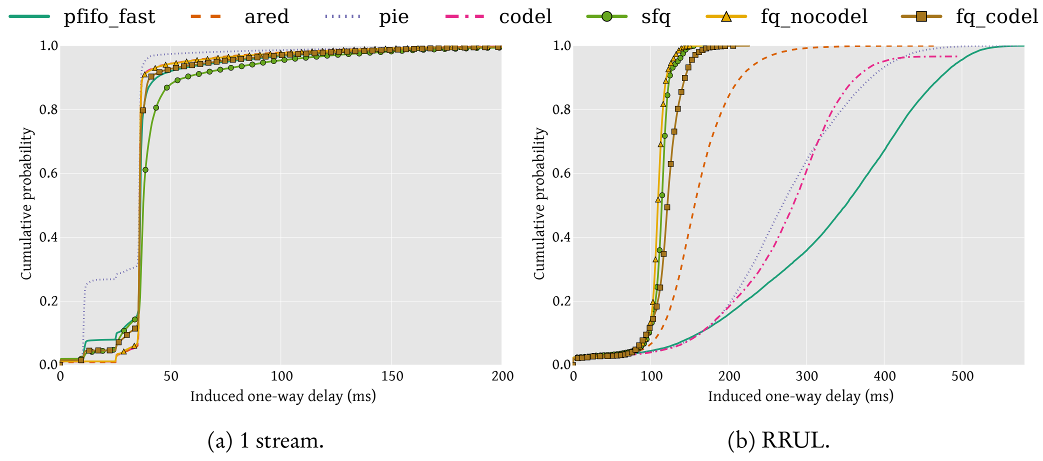
7.4 Discussion

The WiFi results clearly show that the queue management algorithms fail to effectively control the bandwidth on a WiFi bottleneck link. This is most likely due to extra queueing at lower layers in the network stack. Additionally, other issues are apparent with WiFi traffic, most notably the poor bidirectional throughput. It is doubtful that straight-forward solutions exist to these issues, but we believe this to be an interesting avenue for further research. Moreover, in light of the positive results of applying queue management algorithms in general, we believe that they can play a role in solving WiFi's problems as well.

8 Conclusions and future work

We have compared three modern AQM algorithms, revealing three aspects of the AQM behaviour: the Good, the Bad and the WiFi.

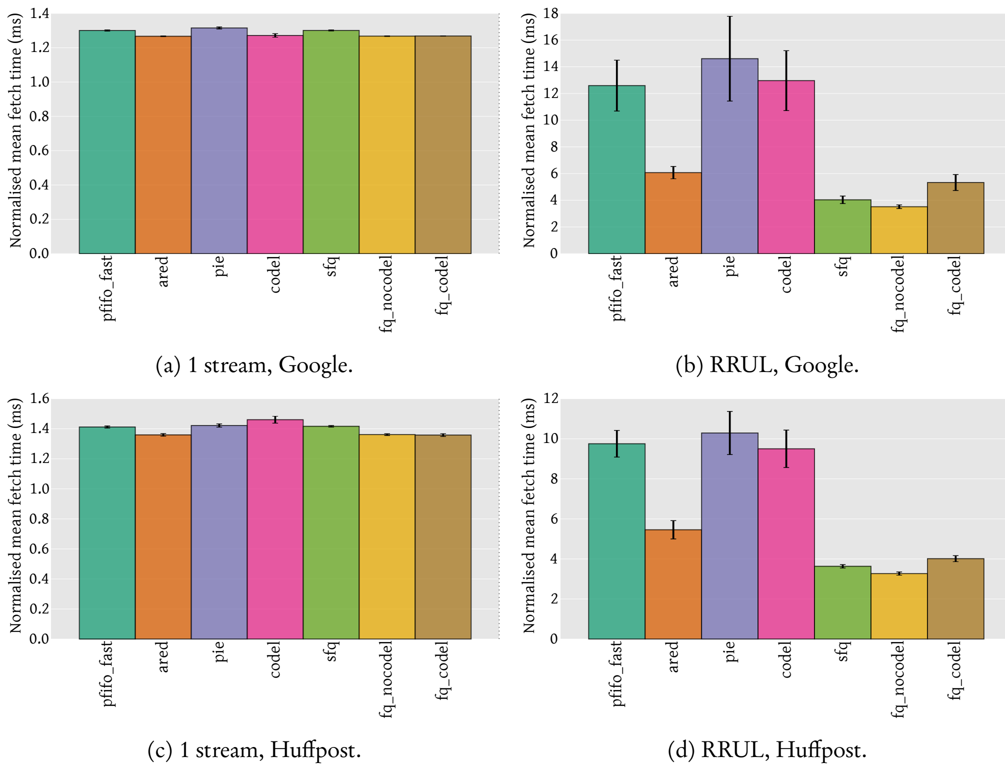
The Good: We show that in the steady state, the new AQM algorithms (PIE and CoDel) show consistent improvements over FIFO queueing, as does the older ARED algorithm. The relative performance of the three algorithms varies with link characteristics; although ARED exhibits a slight tendency to drop too aggressively, hurting throughput but improving latency. This matches previous evaluations well.



(a) 1 stream.

(b) RRUL.

Figure 10: VoIP test results for WiFi.



(a) 1 stream, Google.

(b) RRUL, Google.

(c) 1 stream, Huffpost.

(d) RRUL, Huffpost.

Figure 11: Web test results for WiFi.

The Bad: The fairness results show that the AQM algorithms exacerbate TCP unfairness compared to FIFO queueing. This aspect is often overlooked in evaluations of AQM algorithms, but can be an important factor especially when considering deployment of an AQM algorithm on a link likely to see traffic with highly varying RTT: unfairness can potentially cause flows with long RTTs to suffer degraded throughput, needlessly hurting performance. The examination of transient behaviour shows that the CoDel and PIE algorithms (ARED fares significantly better in this regard) can take several seconds to get delay back under control after a significant load spike occurs, such as the RRUL flow startup; in some cases even performing *worse* than FIFO queueing.

The WiFi: When adding a WiFi link as the bottleneck, we see that all the queue management schemes fail to contain queueing latency. We attribute this to queueing in lower layers of the WiFi stack, and it is clear that more work is needed to properly address this: due to the nature of the physical layer (incorporating retransmissions and packet aggregation features), it is not clear that existing solutions from other media can translate directly to WiFi.

The analysis of these three aspects is an important contribution to understanding AQM behaviour. In particular, the transient behaviour has potential to significantly impact the perceived performance of the network, especially considering that traffic complexity and deployment of highly bursty applications is only increasing. Hence, these types of transient events are likely to be frequent enough that dealing with them needs to be a priority. Likewise, WiFi behaviour is an obvious are of potential improvement.

Our accompanying analysis of the fairness queueing algorithms as a possible remedy for some of the shortcomings of the pure AQM algorithms shows very promising results. The fairness queueing algorithms exhibit steady state goodput and latency generally superior to the AQM algorithms, they ensure almost perfect fairness between flows and they prove to be an effective remedy for transient latency spikes at flow startup. For WiFi, they still suffer from queueing in the lower layers, but perform better than the pure AQMs. One caveat is that the fairness queueing algorithms implicitly enforce sharing and prioritisation constraints between flows that may be unsuitable for some applications and scenarios different from those tested here. However, generally we believe there is a convincing case for fairness queueing algorithms playing an important role in ensuring low latency and high throughput in modern (access) networks.

While the use of better queue management algorithms is proliferating,[12] deployment remains a challenge. And developing comprehensive queue management solutions for different physical layer technologies constitutes important work, which can come with its own challenges, as we have seen in the WiFi example. WiFi in particular remains a challenge (as does other mobile technologies), but getting queue management deployed in places like cable and DSL head-end equipment is also needed.

[12]For instance, fq_codel is the default in the latest versions of the OpenWrt, Fedora and Arch Linux distributions, and PIE will be part of the upcoming DOCSIS 3.1 standard.

Queue management surely plays an important role in ensuring tomorrow's internet provides reliably low-latency connections everywhere, but other technologies also have a role to play, and are developing at a rapid pace. In particular, the Linux networking stack continues to evolve, and in the versions since the 3.14 kernel we have used for our tests, the kernel has seen several tweaks to the TCP stack in particular, along with the inclusion of a whole new congestion control algorithm (DataCenter TCP). Some of these improvements are distinctive in themselves, and some of them have the potential to interact with queue management algorithms in various ways. Figuring out the details of these interactions is also important going forward.

Finally, as we have pointed out in our experiments, the existing queue management schemes are not without issues in certain areas. Most notably, the transient behaviour is an area in need of further study. Together, we consider these issues to be promising potential avenues for further inquiry, and remain optimistic that tomorrow's internet will provide us with reliably low latency at all layers.

References

[1] B. Briscoe *et al.*, "Reducing internet latency: A survey of techniques and their merits," *IEEE Communications Surveys Tutorials*, vol. 18, no. 99, pp. 2149–2196, 2014.

[2] J. Gettys and K. Nichols, "Bufferbloat: Dark buffers in the internet," *ACM Queue*, vol. 9, no. 11, pp. 40–54, Nov. 2011.

[3] C. Staff, "BufferBloat: what's wrong with the internet?" *Communications of the ACM*, vol. 55, no. 2, pp. 40–47, Feb. 2012.

[4] S. Floyd and V. Jacobson, "Random early detection gateways for congestion avoidance," *IEEE/ACM Transactions on Networking*, vol. 1, no. 4, pp. 397–413, 1993.

[5] K. Nichols and V. Jacobson, "Controlling queue delay," *Communications of the ACM*, vol. 55, no. 7, pp. 42–50, Jul. 2012.

[6] R. Pan *et al.*, "PIE: A lightweight control scheme to address the bufferbloat problem," in *2013 IEEE 14th International Conference on High Performance Switching and Routing (HPSR)*, July 2013, pp. 148–155.

[7] S. Floyd, R. Gummadi, and S. Shenker, "Adaptive RED: An algorithm for increasing the robustness of RED's active queue management," 2001. http://www.icir.org/floyd/papers.html

[8] J. Gettys, "Traditional AQM is not enough," Blog post, July 2013. https://gettys.wordpress.com/2013/07/10/low-latency-requires-smart-queuing-traditional-aqm-is-not-enough/

[9] P. McKenney, "Stochastic fairness queueing," in *INFOCOM '90. Ninth Annual Joint Conference of the IEEE Computer and Communication Societies*, vol. 2. IEEE, jun 1990, pp. 733–740.

[10] T. Høiland-Jørgensen *et al.*, "The Flow Queue CoDel Packet Scheduler and Active Queue Management Algorithm," RFC 8290 (Experimental), RFC Editor, Jan. 2018.

[11] R. Adams, "Active Queue Management: A Survey," *IEEE Communications Surveys & Tutorials*, vol. 15, no. 3, pp. 1425–1476, 2013.

[12] F. Checconi, L. Rizzo, and P. Valente, "QFQ: Efficient packet scheduling with tight guarantees," *IEEE/ACM Transactions on Networking (TON)*, vol. 21, no. 3, pp. 802–816, 2013.

[13] A. Kortebi, S. Oueslati, and J. W. Roberts, "Cross-protect: Implicit service differentiation and admission control," in *2004 Workshop on High Performance Switching and Routing*. IEEE, 2004, pp. 56–60.

[14] W.-C. Feng *et al.*, "Stochastic Fair Blue: A Queue Management Algorithm for Enforcing Fairness," in *Proceedings IEEE INFOCOM 2001*. IEEE, 2001, pp. 1520–1529.

[15] G. White, "Active Queue Management in DOCSIS 3.X Cable Modems," Cable Television Laboratories, Inc., Tech. Rep., May 2014. http://www.cablelabs.com/wp-content/uploads/2014/06/DOCSIS-AQM_May2014.pdf

[16] N. Khademi, D. Ros, and M. Welzl, "The new AQM kids on the block: Much ado about nothing?" Oslo University, Tech. Rep. 434, 2013. https://www.duo.uio.no/handle/10852/37381

[17] V. P. Rao, M. P. Tahiliani, and U. K. K. Shenoy, "Analysis of sfqCoDel for Active Queue Management," in *2014 Fifth International Conference on the Applications of Digital Information and Web Technologies (ICADIWT)*. IEEE, 2014, p. 262–267.

[18] I. Järvinen and M. Kojo, "Evaluating CoDel, PIE, and HRED AQM techniques with load transients," in *39th Annual IEEE Conference on Local Computer Networks*. IEEE, 09 2014, pp. 159–167.

[19] K. Cai *et al.*, "Wireless Unfairness: Alleviate MAC Congestion First!" in *Proceedings of the Second ACM International Workshop on Wireless Network Testbeds, Experimental Evaluation and Characterization*. ACM, 2007, pp. 43–50.

[20] G. Park, H. Ko, and S. Pack, "Simulation study of bufferbloat problem on wifi access point," in *2014 IEEE 3rd Global Conference on Consumer Electronics (GCCE)*, Oct 2014, pp. 729–730.

[21] M. Carbone and L. Rizzo, “Dummynet revisited,” *ACM SIGCOMM Computer Communication Review*, vol. 40, no. 2, pp. 12–20, March 2010.

[22] A. N. Kuznetsov, “tbf – Token Bucket Filter,” Linux man page, 2014.

[23] D. Taht and J. Gettys, “Best practices for benchmarking Codel and FQ Codel,” Wiki page on bufferbloat.net web site, September 2014. https://www.bufferbloat.net/projects/codel/wiki/Best_practices_for_benchmarking_Codel_and_FQ_Codel/

[24] D. Taht, “Implementing comprehensive queue management on home routers,” Internet Draft, 2014. http://snapon.lab.bufferbloat.net/ ˜ d/draft-taht-home-gateway-best-practices-00.html

[25] R. Jones, “Netperf,” Open source benchmarking software, 2015. http://www.netperf.org/

[26] A. Botta, A. Dainotti, and A. Pescapè, “A tool for the generation of realistic network workload for emerging networking scenarios,” *Computer Networks*, vol. 56, no. 15, pp. 3531–3547, 2012.

[27] D. Stenberg, “curl and libcurl,” Project web site, 2015. https://curl.haxx.se/

[28] T. Høiland-Jørgensen, “Flent: The FLExible Network Tester,” Project web site, 2015. https://flent.org

[29] G. White and R. Pan, “A PIE-Based AQM for DOCSIS Cable Modems,” Internet Draft (informational), January 2015. https://tools.ietf.org/html/draft-white-aqm-docsis-pie

[30] V. Misra, W.-B. Gong, and D. Towsley, “Fluid-based analysis of a network of AQM routers supporting TCP flows with an application to RED,” *SIGCOMM Comput. Commun. Rev.*, vol. 30, no. 4, pp. 151–160, Aug. 2000.

[31] D. Taht, “Realtime Response Under Load (RRUL) Test,” November 2012. https://www.bufferbloat.net/projects/bloat/wiki/RRUL_Spec/

[32] K. Winstein, “Transport architectures for an evolving internet,” Ph.D. dissertation, Massachusetts Institute of Technology, Jun. 2014. https://cs.stanford.edu/ ˜ keithw/www/Winstein-PhD-Thesis.pdf

[33] H. Balakrishnan and V. N. Padmanabhan, “How network asymmetry affects TCP,” *Communications Magazine, IEEE*, vol. 39, no. 4, pp. 60–67, 2001.

[34] T. Høiland-Jørgensen, “http-getter,” Source code repository, 2014. https://github.com/tohojo/http-getter

[35] A. Kortebi *et al.*, "Evaluating the number of active flows in a scheduler realizing fair statistical bandwidth sharing," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 33. ACM, 2005, pp. 217–228.

[36] J. Padhye *et al.*, "Modeling TCP throughput: A simple model and its empirical validation," in *ACM SIGCOMM Computer Communication Review*, vol. 28, no. 4. ACM, 1998, pp. 303–314.

[37] S. Ha, I. Rhee, and L. Xu, "CUBIC: A new TCP-friendly high-speed TCP variant," *Operation Systems Review*, vol. 42, no. 5, pp. 64–74, Jul. 2008.

[38] T. Kozu, Y. Akiyama, and S. Yamaguchi, "Improving RTT fairness on CUBIC TCP," in *2013 First International Symposium on Computing and Networking (CANDAR)*, Dec 2013, pp. 162–167.

[39] R. Jain, D.-M. Chiu, and W. R. Hawe, *A Quantitative Measure Of Fairness And Discrimination For Resource Allocation In Shared Computer Systems*. Eastern Research Laboratory, Digital Equipment Corporation, 1984.

[40] B. Briscoe, "[aqm] CoDel's control law that determines drop frequency," IETF AQM mailing list message, Nov 2013. https://www.ietf.org/mail-archive/web/aqm/current/msg00376.html

[41] R. Pan, "Re: [aqm] draft-ietf-aqm-pie-01: review," IETF AQM mailing list message, May 2015. https://www.ietf.org/mail-archive/web/aqm/current/msg01216.html

[42] T. Herbert, "bql: Byte Queue Limits," Patch posted to the Linux kernel network development mailing list, Nov 2011. https://lwn.net/Articles/454378/

Paper III

Analysing the Latency of Sparse Flows in the FQ-CoDel Queue Management Algorithm

Reprinted from

IEEE Communication Letters, October 2018

"If we're sticking code into boxes to deploy CoDel, don't do that. Deploy FQ-CoDel. It's just an across the board win."

Van Jacobsson, at IETF 84

Analysing the Latency of Sparse Flows in the FQ-CoDel Queue Management Algorithm

Toke Høiland-Jørgensen
toke.hoiland-jorgensen@kau.se

Abstract

The FQ-CoDel queue management algorithm was recently published as an IETF RFC. It achieves low latency especially for low-volume (or *sparse*) traffic flows competing with bulk flows. However, the exact conditions for when a particular flow is considered to be sparse has not been well-explored.

In this work, we analyse the performance characteristics of the sparse flow optimisation of FQ-CoDel, formulating the constraints that flows must satisfy to be considered sparse in a given scenario. We also formulate expressions for the expected queueing latency for sparse flows.

Then, using a numerical example, we show that for a given link and a given type of sparse flows (VoIP traffic), the number of sparse flows that a given bottleneck can service with low sparse flow latency is only dependent on the number of backlogged bulk flows at the bottleneck. Furthermore, as long as the maximum number of sparse flows is not exceeded, all sparse flows can expect a very low queueing latency through the bottleneck.

1 Introduction

The FQ-CoDel queue management algorithm, which was recently published as an IETF RFC [1], is a hybrid AQM and packet scheduling algorithm that has been shown to be an excellent remedy for the bufferbloat problem of excessive queueing on a congested link [2]. In particular, FQ-CoDel achieves very low latency for low-volume traffic competing with the bulk flows causing the congestion. This is due to the *sparse flow optimisation* employed in the flow scheduler.

However, while FQ-CoDel has been shown to achieve very low latency for such sparse flows, the exact conditions for when a particular flow is considered to be sparse has not been well-explored, as noted in the RFC [1, Section 1.3].

The contribution of this work is an analysis of what exactly constitutes a sparse flow in FQ-CoDel. We achieve this by formulating analytical expressions for the constraints flows must satisfy to be treated as sparse by the FQ-CoDel scheduler, and supplement with a numeric example and simulation for a typical example of real-world traffic (real-time VoIP traffic).

The rest of this paper is structured as follows: Section 2 first summarises related work and Section 3 explains how the sparse flow optimisation in FQ-CoDel works. Section 4 then presents our analytical framework and results and Section 5 shows the real-world examples. Finally, Section 6 concludes.

2 Related work

While several studies have measured the performance of FQ-CoDel (e.g., [2–4]), none deal specifically with the sparse flow optimisation, and none offer any analytical expressions for the performance of the algorithm. However, similar algorithms have been subject to analysis, as summarised below.

FQ-CoDel is based on the deficit round-robin (DRR) scheduler [5]. The authors of DRR propose an extension called DRR+ where "latency-sensitive flows" are given priority as long as such a flow never sends more than x bytes every time period T, which can be said to be an a priori analytical expression for the constraints of a sparse flow. This mechanism is expanded upon in the DRR++ [6] algorithm, which adds an extension to the mechanism to better deal with bursty latency-sensitive flows. The scheduling of DRR++, in particular, is identical to that of FQ-CoDel, except that DRR++ requires flows to be explicitly classified as latency-sensitive (without specifying any mechanism to do so), whereas FQ-CoDel applies the same scheduling to all flows, which means that latency-sensitive flows are only classified implicitly. However, since the authors of DRR++ assume an a priori classification of latency-sensitive flows, there is no analysis of their constraints.

The implicit classification mechanism of FQ-CoDel is similar to that used by the Shortest Queue First (SQF) queueing scheme [7], which works by simply dequeueing packets from the shortest queue available at the time of dequeue. This gives implicit priority to flows that do not build a queue, such as voice flows and low-bandwidth video streams. However, since SQF does not use a round-robin scheduler, it gives no service guarantees to the backlogged bulk flows. The authors provide both analytical and experimental evaluations of the algorithm performance characteristics in [8].

The Quick Fair Queueing (QFQ) algorithm [9] is an O(1) scheduling algorithm that implements fairness queueing between flows in a way that approximates a fluid model of the flows with high accuracy. The paper provides an extensive analysis of its performance characteristics.

Finally, a comprehensive analysis of the number of active flows in a fairness queueing system is provided in [10]. This does not treat queueing latency, nor does it distinguish between types of traffic, such as sparse or bulk flows.

3 The sparse flow optimisation

The FQ-CoDel sparse flow optimisation works as follows:

When a packet arrives at the router, it is hashed on its transport layer 5-tuple (source and destination IP, IP protocol number and source and destination ports). The result of the hash, modulo the number of configured queues, is the queue number of that packet, and the packet is enqueued to that queue. If this queue is already active, no further action is taken. However, if the queue is **not** already active, it is made active by being added to the end of the list of *new queues*.

When dequeueing a packet, FQ-CoDel first finds a queue to dequeue from. This is done by first looking at the list of new queues, which gives priority to queues that recently transitioned from inactive to active. If the list of new queues is empty, a queue is selected from the list of *old queues* (which is every queue that is not a new queue). Having selected the appropriate queue (either new or old), that queue gets to dequeue packets at most totalling *quantum* bytes (which is configurable, but defaults to one MTU), and afterwards the queue is moved to the end of the list of old queues. When a queue becomes empty, it is removed (and so transitions to the inactive state) as long as it has transitioned through the list of old queues at least once.

Since empty queues return to the inactive state, it is possible for a flow to have all its packets trigger the re-activation of the queue when they arrive at the router, which will give the flow effective priority for its entire duration. In the following, we explore what it takes for a flow to achieve this.

4 Analytical framework

Consider an FQ-CoDel instance managing a bottleneck with transmission rate R bytes per second, with N backlogged flows sharing the bottleneck (and so each achieving a rate of R/N bytes per second). We do not concern ourselves with the performance of the N flows, and we assume no hash collisions occur. We furthermore assume all flows transmit packets of equal size L bytes and that the FQ-CoDel quantum $Q = L$.

4.1 One sparse flow

Consider a sparse flow S transmitting packets of size $L_S \leq L$ bytes. What is the maximum transmission rate that permits this flow to be prioritised by the sparse flow mechanism? We first assume that the packets of S are equally spaced with inter-arrival time I_S seconds.

When a packet from flow S arrives at the bottleneck, it will have to wait for the packet currently being serviced to complete transmission. After this, the queue of flow S will be activated as a new queue (i.e., get priority) and be serviced immediately. Once the packet has been transmitted, the queue will be moved to the end of the list of old queues, and if it is still empty after the scheduler has cycled through all the backlogged flows, it will be removed.

This means that to get treated as sparse, the next packet from S has to arrive after the queue has been removed from the scheduler. I.e., after the transmission of the previous packet in S, plus the bulk packet being serviced on arrival, and one additional packet from each backlogged flow. This translates to the following constraints on S:

$$I_S > \frac{L(N+1) + L_S}{R} \Rightarrow R_S < \frac{R}{\frac{L}{L_S}(N+1) + 1} \tag{1}$$

Where R_S is the rate of flow S.

Next, we consider what happens if the packets of S are not equally spaced (i.e., that I_S varies between subsequent packets), but still obeys the rate restriction in (1). There are two cases to consider: The case where the packets of S are sent in *bursts* of several back-to-back packets with longer spaces between them, and the case where the inter-arrival time simply varies so that, say, every other packet pair obeys (1) and every other pair does not.

In the case of bursts we assume that the bursts themselves are equally spaced over the lifetime of the flow. If the total burst size is less than the quantum size (i.e., $Q >= nL_S$ for bursts of n packets), all packets in the burst will be dequeued at the same time, and we can simply consider the behaviour equivalent to the case where the flow consists of single equally spaced packets of size nL_S. If the burst is larger than the quantum size, the first Q bytes of each burst will be dequeued immediately, while the rest will be queued until the next round of the scheduler[13].

For the non-burst case, we consider the packets $p_1, \ldots, p_n$ of flow S with inter-arrival times $i_1, \ldots, i_{n-1}$ since the previous packet. By assumption, the average inter-arrival time is I_S and obeys (1). This means that inter-arrival times will alternate between being less than or more than I_S. I.e., every sequence of consecutive packet arrivals with inter-arrival times $i_0^-, \ldots, i_j^- < I_S$ will be followed by a sequence of packet arrivals with inter-arrival times $i_0^+, \ldots, i_k^+ > I_S$ (otherwise (1) wouldn't hold). We label the i'th sequence of packets with inter-arrival times $< I_S$ as I_i^-, and the (ordered) set of all such sequences as $\mathbf{I}^-$. Similarly, the j'th sequence of packets with inter-arrival times $>= I_S$ are labelled I_j^+, with the set of all such sequences given as $\mathbf{I}^+$. We furthermore impose a *regularity constraint* on the flow:

$$\forall I_i^- \in \mathbf{I}^- : \frac{\overline{I_i^-} + \overline{I_i^+}}{2} \geq I_S \tag{2}$$

where $\overline{I_i^-}$ is the average value of $i_k \in I_i^-$. I.e., (2) states that every sequence of packets with inter-arrival times smaller than I_S must be followed by a sequence of packets with inter-arrival times larger than I_S, such that the average inter-arrival time satisfies (1) when looking only at those two sub-sequences.

Given these constraints, packets in I^+ will receive the low latency performance from the sparse flow optimisations, while packets in I^- will arrive while

[13] Since we assume that the average rate of the flow obeys (1), the queue has to be cleared out before the next burst.

the queue is already being scheduled, and so will experience higher queueing latency. The actual queueing latency experienced by packets in I^- depends on the distribution of packets; exploring this is out of scope for this analysis.

4.2 Multiple sparse flows

If M sparse flows go through the bottleneck, and we assume that all sparse flows have the same packet inter-arrival time I_S, this inter-arrival time will have to satisfy:

$$I_S > \frac{L(N+1) + L_S M}{R} \tag{3}$$

In the worst case scenario, the sparse flows synchronise (so packets from all flows arrive at the same time). In this case, the expected queueing latency for all sparse flows will be:

$$\frac{L_S(M-1) + L}{2R} \tag{4}$$

Where the L is due to the bulk flows not being preempted.

However, this worst-case latency is only seen if the sparse flows synchronise so that their packets arrive at the same time (and have to queue behind one another). We can express expected queueing latency of a sparse flow in the average case by modelling the arrivals of sparse flows as follows.

Since we have bounded the inter-arrival time for each flow by (3), all flows are by assumption sparse themselves. This means that when a packet on a given sparse flow arrives at the bottleneck, it will not queue behind any other packets from the same flow, but only behind other sparse flows. Since the sparse flows are served in round-robin order, this becomes equivalent to a FIFO queue of flows waiting to be serviced (each of which has a single packet queued), and so we are really expressing the distribution of flow start times. Assuming Poisson arrivals for the flows, this system can be expressed as an $M/D/1$ queue (as link capacity and packet sizes are fixed). This will allow us to express an upper bound on the expected queueing latency of the sparse flows (since the flow arrival distribution with a fixed number of flows would be a right-truncated exponential distribution, rather than the exponential distribution assumed in an $M/D/1$ setting).

This $M/D/1$ queueing system has the following values for arrival rate (λ), service rate (μ) and utilisation (ρ):

$$\lambda = M/I_S, \quad \mu = R/L_S, \quad \rho = \frac{ML_S}{RI_S} \tag{5}$$

From this, we can straight-forwardly express the expected queueing time ω_q as a function of the number of sparse flows, the packet size and inter-arrival times and the link rate:

$$\omega_q = \frac{\rho}{2\mu(1-\rho)} = \frac{\frac{ML_S}{RI_S}}{2\frac{R}{L_S}\left(1-\frac{ML_S}{RI_S}\right)} = \frac{ML_S}{2\frac{R}{L_S}\left(RI_S - ML_S\right)} \tag{6}$$

This is useful for predicting and upper bound the expected queueing time for any concrete flow type (where these values are known), as we will see in Section 5. Note that in the case where there are also bulk flows present, we need to add $L/2R$ to the expected queueing time, to account for the packet that is being processed when a packet from a sparse flow arrives.

4.3 Impact on bulk flows

Since the sparse flow optimisation simply corresponds to inserting new queues at the head of the round-robin list instead of at the tail, the steady-state performance impact on bulk flows is the same as for DRR; i.e., given two flows flow i and j, for each dequeue opportunity afforded to i, j has at least one dequeue opportunity. As such, the expected service given to each flow scales with the total number of bulk and sparse flows (i.e., it is proportional to $N + M$), in the worst case. In practice, many sparse flows will have rates significantly lower than the bulk flows, in which case the DRR scheduler will divide the spare capacity between the backlogged bulk flows.

4.4 Impact of changing the quantum

We initially assumed that the quantum $Q = L$, which means that a bulk flow can dequeue a full packet every time it is scheduled. If $Q > L$, every bulk flow is still serviced every scheduling round, but may dequeue more than one packet. Whereas, if $Q < L$, each bulk flow will get a dequeue opportunity every L/Q scheduling rounds and, conversely, only QN/L bulk flows will dequeue a packet each round. In the case where only bulk flows are present, these two effects cancel each other out. However, in the presence of sparse flows, the quantum impacts the bounds on sparse flow inter-arrival time given in (3). Assuming $Q >= L_S$ so sparse flows always dequeue a full packet when they are scheduled, this becomes:

$$I_S > \frac{Q(N+1) + L_S M}{R} \tag{7}$$

5 Real-world examples

Using (6) and (7) we can compute two useful properties: The maximum number of sparse flows a given link can sustain as a function of the number of backlogged bulk flows at the bottleneck, and the expected queueing time

for each such sparse flow. To do this, we can rewrite (7) as follows, while also including the case where there are no bulk flows:

$$M_{max} < \begin{cases} \dfrac{I_S R}{L_S}, & \text{if } N = 0 \\ \dfrac{I_S R - Q(N+1)}{L_S}, & \text{if } N > 0 \end{cases} \tag{8}$$

We also need to assign some values to the variables in the equations. For our example, we consider a 10 Mbps Ethernet link where bulk flows transmit full-size (1518 bytes) packets and the quantum is set to coincide with this full packet size (as is the default in FQ-CoDel), and the sparse flows consist of a number of G.711 VoIP flows at the highest (64 Kbps) data rate, which transmits packets of 218 bytes (160 bytes payload + RTP, UDP, IP and Ethernet headers) at a fixed 20 ms interval. These values are summarised in Table 1.

Table 1: Values used in the numerical example

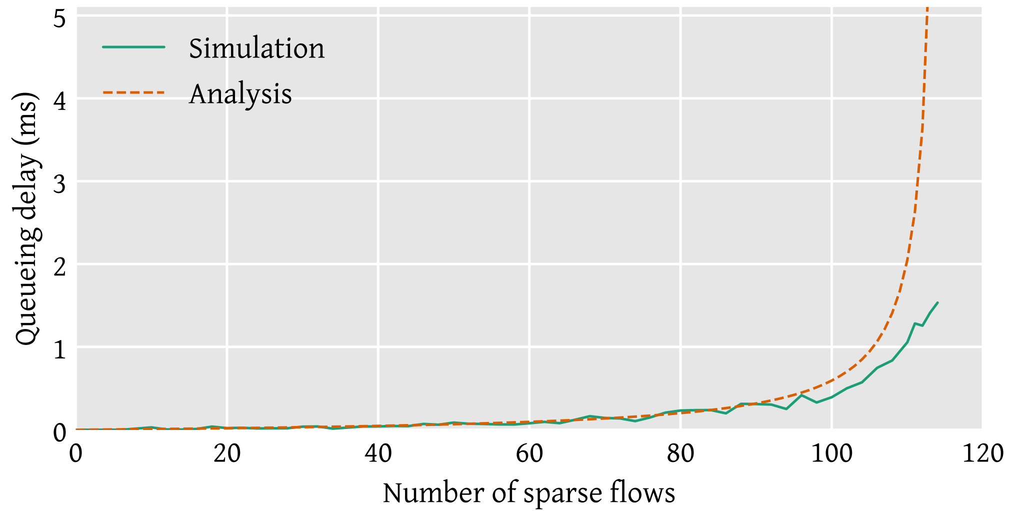
Variable	Value
Q	$1518B$
L_S	$218B$
I_S	$0.02s$
R	1.25 MB/s (10 Mbps)

With these values, (8) tells us that the maximum number of VoIP flows the bottleneck can handle while still treating them as sparse flows, is a linear function of the number of bulk flows backlogged at the bottleneck. With no bulk flows, 114 sparse flows can be serviced, which correspond to the number of VoIP flows the bottleneck link has capacity for (each flow transmits at a link-level rate of 87.2 Kbps). With 15 bulk flows backlogged, only three simultaneous VoIP flows can traverse the bottleneck link as sparse flows, and with more backlogged flows, the VoIP flows will no longer be treated as sparse. The number of sparse flows per bulk flow is related to the ratio between the quantum and the packet size of the VoIP flows.

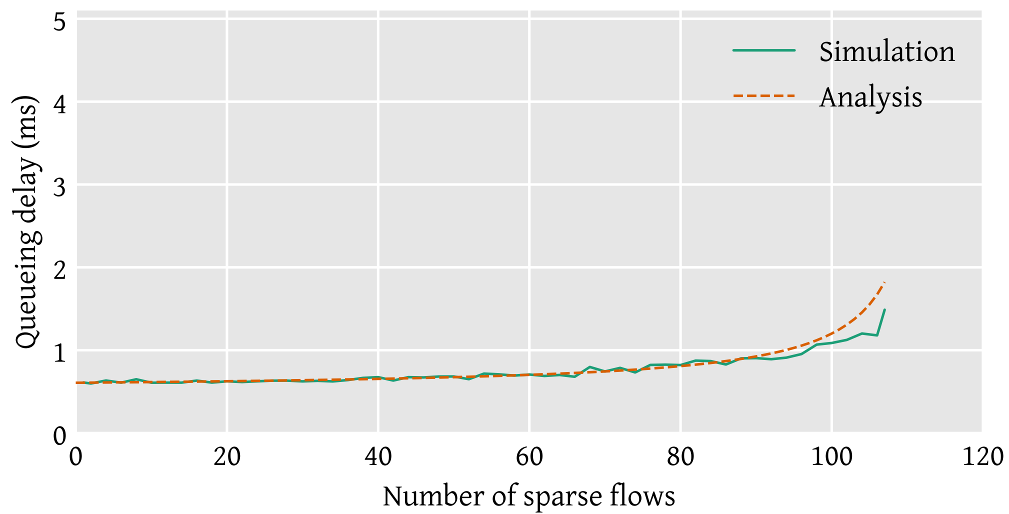
Turning to the expected queueing time of the sparse flows themselves, Figure 1 shows this as a function of the number of sparse flows. To verify that the model accurately predicts an upper bound on the queueing latency, we have also created a simulation of FQ-CoDel in the Salabim event-driven simulator[14]. The results from the simulation are also included in the figure.[15] Note that the expected queueing time does not depend on the number of bulk flows, other than to limit the number of sparse flows that can be supported. In fact, a sparse flow can experience lower latency when competing against

[14] `http://www.salabim.org`

[15] The bulk flows used in the simulation are fixed rate UDP flows. Further details of the simulation runs are omitted here due to space constrains, but are available (along with the full simulation source code) in [11].



(a) No bulk flows



(b) One bulk flow

Figure 1: Expected queueing delay as a function of the number of sparse flows at the bottleneck.

bulk flows, than when competing against a large number of other sparse flows. This limiting is illustrated by the two graphs, where Figure 1a shows the case of no bulk flows and Figure 1b shows the case of a single bulk flow.

The thing to note here is that the expected queueing latency is kept very low for all the sparse flows, and that adding bulk flows does not change this, other than to add a constant to the queueing time corresponding to the packet being processed when a sparse flow packet arrives. In fact, as Figure 1b shows, the expected queueing latency even for the maximum number of sparse flows that the link can handle, is less than two milliseconds with one or more bulk flows limiting the number of sparse flows. So as long as an operator is using (8) to calculate the max number of sparse flows the link can support, she can be confident that the sparse flows themselves will achieve very low queueing latency at the bottleneck.

6 Conclusion

We have analysed the performance characteristics of the sparse flow optimisation of FQ-CoDel. This analysis shows the constraints that flows must satisfy to be considered sparse in a given scenario, which is dependent on the number of flows (both bulk and sparse) and the link rate. We also formulate expressions for the expected queueing latency for sparse flows.

Using a numerical example, we also show that for a given link and a given type of sparse flows (VoIP traffic), the number of sparse flows that a given bottleneck can service with the low sparse flow latency is only dependent on the number of backlogged bulk flows at the bottleneck. And that as long as the maximum number of sparse flows is not exceeded, all sparse flows can expect a very low queueing latency.

7 Acknowledgements

Many thanks to Daniel Larsson and Martin Wahlberg for implementing and running the simulation and to Dave Taht, Johan Garcia, Per Hurtig and Anna Brunstrom for feedback on the ideas and for reviewing various versions of the manuscript.

References

[1] T. Høiland-Jørgensen *et al.*, "The Flow Queue CoDel Packet Scheduler and Active Queue Management Algorithm," RFC 8290 (Experimental), RFC Editor, Jan. 2018.

[2] T. Høiland-Jørgensen, P. Hurtig, and A. Brunstrom, "The Good, the Bad and the WiFi: Modern AQMs in a residential setting," *Computer Networks*, vol. 89, pp. 90–106, Oct. 2015.

[3] N. Khademi, D. Ros, and M. Welzl, "The new AQM kids on the block: Much ado about nothing?" Oslo University, Tech. Rep. 434, 2013.

[4] J. Kua, G. Armitage, and P. Branch, "The impact of active queue management on dash-based content delivery," in *IEEE 41st Conference on Local Computer Networks (LCN)*, Nov 2016, pp. 121–128.

[5] M. Shreedhar and G. Varghese, "Efficient fair queuing using deficit round-robin," *IEEE/ACM Transactions on Networking*, Jun. 1996.

[6] M. H. MacGregor and W. Shi, "Deficits for bursty latency-critical flows: DRR++," in *IEEE International Conference on Networks, 2000.* IEEE, 2000, pp. 287–293.

[7] T. Bonald, L. Muscariello, and N. Ostallo, "Self-prioritization of audio and video traffic," in *2011 IEEE International Conference on Communications (ICC).* IEEE, 2011.

[8] G. Carofiglio and L. Muscariello, "On the impact of TCP and per-flow scheduling on internet performance," *IEEE/ACM Transactions on Networking*, vol. 20, no. 2, pp. 620–633, 2012.

[9] F. Checconi, L. Rizzo, and P. Valente, "QFQ: Efficient packet scheduling with tight guarantees," *IEEE/ACM Transactions on Networking*, vol. 21, no. 3, pp. 802–816, 2013.

[10] A. Kortebi *et al.*, "Evaluating the number of active flows in a scheduler realizing fair statistical bandwidth sharing," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 33. ACM, 2005, pp. 217–228.

[11] T. Høiland-Jørgensen, D. Larsson, and M. Wahlberg, "FQ-CoDel Queue Management Algorithm - Sparse Flow Analysis Software," Sep. 2018. https://doi.org/10.5281/zenodo.1420467

Paper IV

Piece of CAKE

A Comprehensive Queue Management Solution for Home Gateways

Reprinted from

IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN 2018), 25–27 June 2018, Washington, DC

"Cake and grief counselling will be available at the conclusion of the test."

GLaDOS, the Portal game

Piece of CAKE

A Comprehensive Queue Management Solution for Home Gateways

Toke Høiland-Jørgensen, Dave Täht and Jonathan Morton
toke.hoiland-jorgensen@kau.se, dave.taht@gmail.com,
chromatix99@gmail.com

Abstract

The last several years has seen a renewed interest in smart queue management to curb excessive network queueing delay, as people have realised the prevalence of bufferbloat in real networks.

However, for an effective deployment at today's last mile connections, an improved queueing algorithm is not enough in itself, as often the bottleneck queue is situated in legacy systems that cannot be upgraded. In addition, features such as per-user fairness and the ability to de-prioritise background traffic are often desirable in a home gateway.

In this paper we present Common Applications Kept Enhanced (CAKE), a *comprehensive network queue management system* designed specifically for home internet gateways. CAKE packs several compelling features into an integrated solution, thus easing deployment. These features include: bandwidth shaping with overhead compensation for various link layers; reasonable DiffServ handling; improved flow hashing with both per-flow and per-host queueing fairness; and filtering of TCP ACKs.

Our evaluation shows that these features offer compelling advantages, and that CAKE has the potential to significantly improve performance of last-mile internet connections.

1 Introduction

Eliminating bufferbloat has been recognised as an important component in ensuring acceptable performance of internet connections, especially as applications and users demand ever lower latencies. The last several years have established that Active Queue Management and Fairness Queueing are effective solutions to the bufferbloat problem, and several algorithms have been proposed and evaluated (e.g., [1–3]).

However, while modern queueing algorithms can effectively control bufferbloat, effective deployment presents significant challenges. The most immediate challenge is that the home gateway device is often not directly in control of the bottleneck link, because queueing persists in drivers or firmware of devices that cannot be upgraded [1]. In addition, other desirable features in a home networking context (such as per-user fairness, or the ability to explicitly de-prioritise background applications) can be challenging to integrate with existing queueing solutions. To improve upon this situation, we have developed Common Applications Kept Enhanced (CAKE), which is a *comprehensive network queue management* system designed specifically for the home router use case.

As outlined below, each of the issues that CAKE is designed to handle has been addressed separately before. As such, the compelling benefit of CAKE is that it takes state of the art solutions and integrates them to provide:

- a high-precision rate-based bandwidth shaper that includes overhead and link layer compensation features for various link types.
- a state of the art fairness queueing scheme that simultaneously provides both host and flow isolation.
- a Differentiated Services (DiffServ) prioritisation scheme with rate limiting of high-priority flows and work-conserving bandwidth borrowing behaviour.
- TCP ACK filtering that increases achievable throughput on highly asymmetrical links.

CAKE is implemented as a *queueing discipline* (qdisc) for the Linux kernel. It has been deployed as part of the OpenWrt router firmware for the last several years and is in the process of being submitted for inclusion in the mainline Linux kernel.[16]

The rest of this paper describes the design and implementation of CAKE and is organised as follows: Section 2 outlines the desirable features of a comprehensive queue management system for a home router, and recounts related work in this space. Section 3 describes the design and implementation of CAKE in more detail, and Section 4 evaluates the performance of the various features. Finally, Section 5 concludes.

2 Background and Related Work

As mentioned initially, CAKE is designed to run on a home network gateway. We have gathered significant experience with implementing such a system in form of the Smart Queue Management (SQM) system shipped in the OpenWrt router firmware project, which has guided the design of CAKE.

[16]We include links to the source code, along with the full evaluation dataset, in an online appendix [4].

In this section we provide an overview of the problems CAKE is designed to address. We are not aware of any previous work addressing the home gateway queue management challenges as a whole. However, several of the issues that CAKE addresses have been subject of previous work, and so the following subsections serve as both an introduction to the design space and an overview of related work.

The four problems we seek to address are bandwidth shaping, queue management and fairness, DiffServ handling and TCP ACK filtering. These are each treated in turn in the following sections.

2.1 Bandwidth Shaping

A queue management algorithm is only effective if it is in control of the bottleneck queue. Thus, queueing in lower layers needs to be eliminated, which has been achieved in Linux for Ethernet [5] and WiFi [6]. However, eliminating queueing at the link layer is not always possible, either because the driver source code is unavailable, or because the link-layer is implemented in inaccessible hardware or firmware (either on the same device or a separate device, such as a DSL modem).

As an alternative, queueing in the lower layers can be avoided by deploying a bandwidth shaper as part of the queue management system. By limiting the traffic traversing the bottleneck link to a bandwidth that is slightly less than the physical capacity of the link itself, queueing at the physical bottleneck can be eliminated and bufferbloat avoided. Such bandwidth shaping can be performed by a token bucket-based shaper (as is well-known from ATM networks, e.g., [7]), or by a rate-based shaper (which is known from video streaming applications, e.g., [8]).

The use of a shaper to move the link bottleneck wastes the bandwidth that is the difference between the actual physical link capacity, and the set-point of the shaper. To limit this waste, the shaper needs to be set as close to the actual link bandwidth as possible, while avoiding sending bursts of packets at a rate that is higher than the actual capacity. To achieve this, accurate timing information on a per-packet basis is needed. In addition, the shaper must account for link-layer framing and overhead. For instance, DSL links using ATM framing split up data packets into an integer number of fixed-size cells, which means that the framing overhead is a step function of packet size, rather than a fixed value.

2.2 Queue Management

Having control of the bottleneck queue makes it possible to implement effective queue management that can all but eliminate bufferbloat. Such a queue management scheme usually takes the form of an Active Queue Management (AQM) algorithm, combined with a form of fairness queueing (FQ). Several such schemes exist, and extensive evaluation is available in the literature (e.g., [1–3, 9–11]).

Among the state of the art algorithms in modern queue management, is the FQ-CoDel algorithm [12]. FQ-CoDel implements a hybrid AQM/fairness queueing scheme which isolates flows using a hashing scheme and schedules them using a Deficit Round-Robin (DRR) [13] scheduler. In addition, FQ-CoDel contains an optimisation that provides implicit service differentiation for sparse (low-bandwidth) flows, similar to [14, 15]. Evaluations of FQ-CoDel have shown that it achieves low queueing latency and high utilisation under a variety of scenarios [1, 3].

However, while the FQ-CoDel scheduler provides flow isolation and fairness, the transport layer flow is not always the right level of fairness in the home gateway use case. Often, additional isolation between *hosts* on the network is desirable; and indeed this per-host isolation was the most requested feature of the SQM system. Host isolation is straight-forward to implement in place of flow fairness in any fairness queueing based scheme (by simply changing the function that maps packets into different queues), but we are not aware of any practical schemes prior to CAKE that implement *both* host and flow fairness.

2.3 DiffServ Handling

Even though flow-based fairness queueing offers a large degree of separation between traffic flows, it can still be desirable to explicitly treat some traffic as higher priority, and to have the ability to mark other traffic as low priority. Since a home network generally does not feature any admission control, any prioritisation scheme needs to be robust against attempts at abuse (so, e.g., a strict priority queue does not work well). In addition, enabling prioritisation should not affect the total available bandwidth in the absence of marked traffic, as that is likely to cause users to turn the feature off.

Prioritisation of different traffic classes can be performed by reacting to DiffServ markings [16]. This is commonly used in WiFi networks, where DiffServ code points map traffic into four priority levels [17]. For the home gateway use case, various schemes have been proposed in the literature (e.g., [18]), but as far as we are aware, none have seen significant deployment.

2.4 TCP ACK Filtering

TCP ACK filtering is an optimisation that has seen some popularity in highly asymmetrical networks [19], and especially in cable modem deployments [20]. The technique involves filtering (or *thinning*) TCP acknowledgement (ACK) packets by inspecting queues and dropping ACKs if a TCP flow has several consecutive ACKs queued. This can improve performance on highly asymmetrical links, where the reverse path does not have sufficient capacity to transport the ACKs produced by the forward path TCP flow. However, ACK filtering can also have detrimental effects on performance, for instance due to cross layer interactions [21].

3 The Design of CAKE

The design of CAKE builds upon the basic fairness scheduler design of FQ-CoDel, but adds features to tackle the areas outlined in the previous section. The following sections outline how CAKE implements each of these features.

3.1 Bandwidth Shaping

CAKE implements a rate-based shaper, which works by scheduling packet transmission at precise intervals using a virtual transmission clock. The clock is initialised by the first packet to arrive at an empty queue, and thereafter is incremented by the calculated serialisation delay of each transmitted packet. Packets are delayed until the system time has caught up with the virtual clock. If the clock schedule is reached while the queue is empty, the clock is reset and the link goes idle.

This shaper handles bandwidth ranging over several orders of magnitude, from several Kbps to several Gbps. In addition, the rate-based shaper does not require a burst parameter, which simplifies configuration as compared to a token-bucket shaper. It also eliminates the initial burst observed from token-bucket shapers after an idle period. This is important for controlling the bottleneck queue, as this initial burst would result in queueing at the real bottleneck link.

3.1.1 Overhead and Framing Compensation

As mentioned in Section 2.1 above, the shaper accounts for the actual size of a packet on the wire, including any encapsulation and overhead, which allows the rate to be set closer to the actual bottleneck bandwidth, thus eliminating waste. We believe it is safe to set a rate within 0.1% of the actual link rate when the overhead compensation is configured correctly, with a margin mainly required to accommodate slight variations in the actual bottleneck link bandwidth, caused by, e.g., clock drift in the hardware.

CAKE implements an overhead compensation algorithm which begins by determining the size of the network-layer packet, stripped of any MAC layer encapsulation. Having determined the network-layer packet size, the configured overhead can be added to yield the correct on-the-wire packet size, followed optionally by a specialised adjustment for ATM or PTM framing. This algorithm is shown in Algorithm 1.

Using the network-layer packet size and adding a manually configured overhead value is required because the values reported by the kernel are often wrong due to idiosyncrasies of the CPE unit. While this does make configuration a bit more complex, we seek to alleviate this by providing keywords for commonly used configurations.

As part of the overhead compensation, CAKE also optionally splits "super packets" generated by hardware offload features. These super packets are essential for operating at high bandwidths, as they help amortise fixed network stack costs over several packets. However, at lower bandwidths they can hurt

Algorithm 1 Shaping and overhead compensation algorithm. *T_next* is the time at which the next packet is eligible for tranmission.

```
1: function ENQUEUE(pkt)
2:     net_len ← pkt.len − NETWORK_OFFSET(pkt)
3:     adj_len ← net_len + overhead
4:     if ATM framing is enabled then
5:         adj_len ← CEILING(adj_len / 48) * 53
6:     else if PTM framing is enabled then
7:         adj_len ← CEILING(adj_len / 64) * 65
8:     pkt.adj_len ← adj_len
9:     if backlog is zero and T_next is after Now then
10:        T_next ← Now
11: function DEQUEUE
12:     if T_next is after Now then
13:         Schedule interrupt at T_next
14:         return Nil
15:     pkt ← Choose Packet
16:     T_next ← T_next + pkt.adj_len ∗ time_per_byte
17:     return pkt
```

latency, in particular when a link with a high physical bandwidth is shaped to a lower rate. For this reason, we conditionally split super packets when shaping at rates lower than 1 Gbps. This allows CAKE to ensure low latency at lower rates, while still scaling to full line rate on a 40Gbps link.

3.2 Flow Isolation and Hashing

CAKE replaces the direct hash function used in FQ-CoDel with an 8-way set-associative hash. While set-associative hashing has been well-known for decades as a means to improve the performance of CPU caches [22], it has not seen much use in packet scheduling. Conceptually, a $k-$way set-associative hash with n total buckets can be thought of as a plain hash with n/k buckets that is only considered to have a collision if more than k items hash into the same bucket. As can be seen in Figure 1, this significantly reduces the hash collision probability up to the point where the number of flows is larger than the number of queues.[17]

3.2.1 Host Isolation

With flow fairness, hosts can increase their share of the available bandwidth by splitting their traffic over multiple flows. This can be prevented by providing host fairness at the endpoint IP address level, which CAKE can do in addition to flow fairness.

The host isolation is simple in concept: The effective DRR quantum is divided by the number of flows active for the flow endpoint. This mechanism

[17] See how we computed these probabilities in the online appendix.

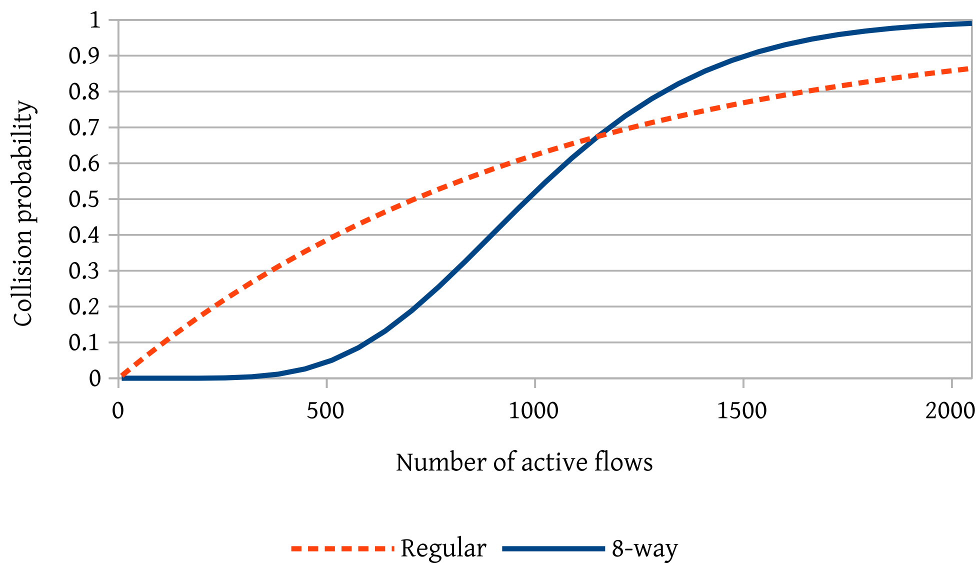


Figure 1: Probability that a new flow will experience a hash collision, as a function of the number of active flows. 1024 total queues.

can be activated in three different modes: source address fairness, in which hosts on the local LAN receive equal share, destination address fairness, in which servers on the public internet receive an equal share, or "triple isolate" mode, in which the maximum of the source and destination scaling is applied to each flow. CAKE also hooks into the Linux kernel Network Address Translation (NAT) subsystem to obtain the internal host address of a packet, which would otherwise be obscured since packets are queued after NAT is applied.

CAKE accomplishes this scaling as shown in Algorithm 2: When a packet is enqueued it is hashed into a queue using the transport layer port numbers along with the source and destination IP addresses. In addition, two separate hashes are performed on the packet destination IP address and source IP address. A separate set of hash buckets is kept for these address hashes. These buckets do not contain a queue of packets, but instead a data structure that keeps two reference counts for each IP address, which track the number of active flows with the given address as source and destination, respectively.

The per-IP reference counts are used to modify the quantum for each active flow. When a flow is scheduled, its "host load" is calculated as the maximum of the reference counts for its source and destination IP addresses. The effective quantum of the flow is simply divided by this load value, which achieves the desired scaling.

3.3 DiffServ handling

CAKE provides a small number of preset configurations, which map each DiffServ code point into a priority tier. If the shaper is in use, each priority tier gets its own virtual clock, which limits that tier's rate to a fraction of the

Algorithm 2 Host isolation algorithm.

```
1:  function ENQUEUE(pkt)
2:      flow_hash ← HASH(pkt.hdr)
3:      src_hash ← HASH(pkt.src_ip)
4:      dst_hash ← HASH(pkt.dst_ip)
5:      flow ← flows[flow_hash]
6:      if flow is not active then
7:          hosts[src_hash].refcnt_src++
8:          hosts[dst_hash].refcnt_dst++
9:          flow.active ← 1
10:         flow.src_id ← src_hash
11:         flow.dst_id ← dst_hash
12: function GET_QUANTUM(flow)
13:     refcnt_src ← hosts[flow.src_id].refcnt_src
14:     refcnt_dst ← hosts[flow.dst_id].refcnt_dst
15:     host_load ← MAX(refcnt_src, refcnt_dst, 1)
16:     return flow.quantum/host_load
```

overall shaped rate. When dequeueing a packet, the algorithm simply picks the highest-priority tier which both has queued traffic and whose schedule is due, if one exists. To allow tiers to borrow excess bandwidth from one another, the dequeue algorithm also tracks the earliest schedule time of all non-empty tiers, and if no other eligible tier is available, that tier is picked instead (within the overall shaper limits).

When the shaper is not in use, CAKE instead uses a simple weighted DRR mechanism to schedule the different priority tiers, with the same weights as the shaper fractions mentioned above. This has weaker precedence guarantees for high-priority traffic, but provides the same proportional capacity reservation and the ability to borrow spare capacity from less than fully loaded tiers.

CAKE defaults to a simple, three-tier mode that interprets most code points as "best effort", but places CS1 traffic into a low-priority "bulk" tier which is assigned 1/16 of the total rate, and a few code points indicating latency-sensitive or control traffic (specifically TOS4, VA, EF, CS6, CS7) into a "latency sensitive" high-priority tier, which is assigned 1/4 rate. The other DiffServ modes supported by CAKE are a 4-tier mode matching the 802.11e precedence rules [17], as well as two 8-tier modes, one of which implements strict precedence of the eight priority levels.

3.4 ACK filtering

CAKE contains an ACK filtering mechanism that drops *redundant* ACKs from a TCP flow. The mechanism takes advantage of the per-flow queueing by scanning the queue after every packet enqueue, to identify a pure ACK (i.e., an ACK with no data) that was made redundant by the newly enqueued packet. An ACK is only filtered if the newly enqueued packet contains an acknowledgement of *strictly more* bytes than the one being filtered. In

particular, this means that duplicate ACKs are not filtered, so TCP's fast retransmit mechanism is not affected. In addition, the filter parses TCP headers and only drops a packet if that will not result in loss of information at the sender; and packets with unknown headers are never dropped, to avoid breaking future TCP extensions. The filter has two modes of operation: a conservative mode that will always keep at least two redundant ACKs queued, and an aggressive mode, that only keeps the most recently enqueued ACK.

4 Performance Evaluation

In this section, we present a performance evaluation of CAKE. All tests are performed on a testbed that emulates a pair of hosts communicating through a low-bandwidth link. We use the Flent testing tool [23] to run the tests, and the data files are available on the companion web site.[16] Unless otherwise stated below, all tests are run on a symmetrical 10 Mbps link with 50 ms baseline latency. Our basic test is the Real-Time Response Under Load test, which consists of running four TCP flows in each traffic direction, along with three different latency measurement flows [24].

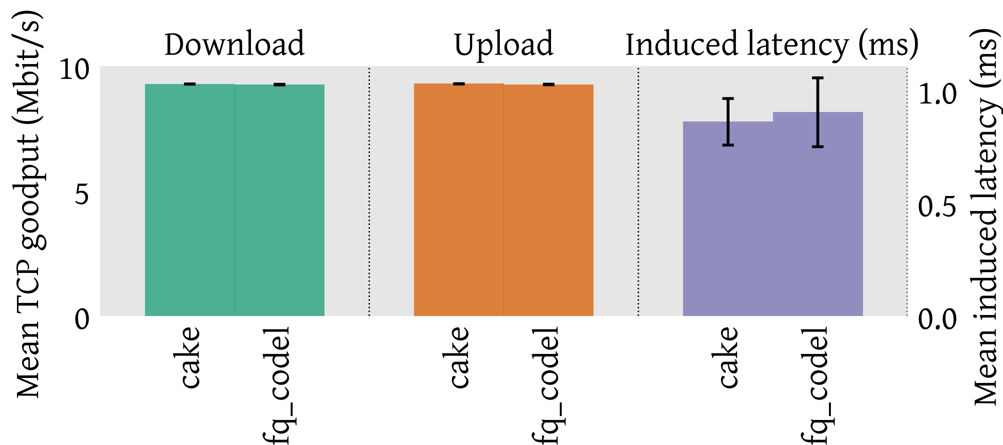


Figure 2: Baseline throughput and latency of CAKE and FQ-CoDel on a 10 Mbps link.

As can be seen in Figure 2, the baseline performance of CAKE is comparable to that of FQ-CoDel: both achieve low latency and high throughput in the baseline test. This is expected, since CAKE is derived from FQ-CoDel. For a more comprehensive comparison of FQ-CoDel with other queue management algorithms, we refer the reader to [1]. Instead, the remainder of this evaluation focuses on the features outlined in the previous sections.

4.1 Host Isolation

To evaluate the host isolation feature of CAKE, we run a varying number of TCP flows between two source hosts and four destination hosts. Source host A runs one flow to each of destination hosts A and B, and two flows to

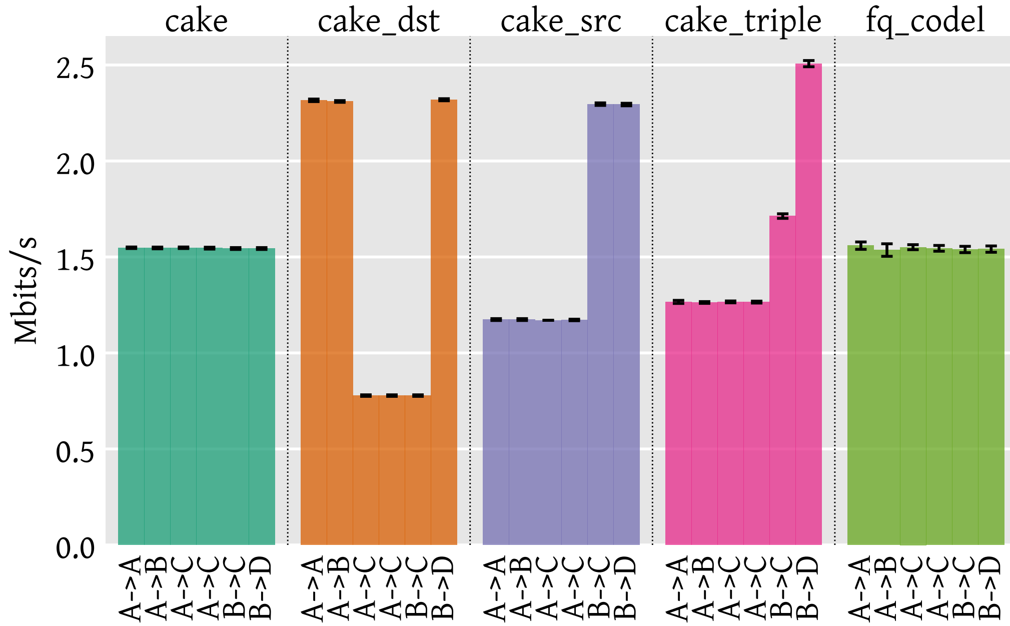


Figure 3: Host isolation performance with TCP flows from two source hosts to four destination hosts. The columns show different algorithms; each bar shows the average flow goodput.

destination host C, while source host B runs one flow to each of destination hosts C and D. This makes it possible to demonstrate the various working modes of CAKE's host isolation feature.

The result of this test is shown in Figure 3. It shows four configurations of CAKE (no host isolation, source host isolation, destination host isolation and triple isolation) and a test with FQ-CoDel as the queue management algorithm. As can be seen in the figure, both FQ-CoDel and CAKE with no host isolation provide complete fairness between all six flows.

The figure also clearly shows the various modes of flow isolation supported by CAKE: In destination fairness mode (second column), the four destination hosts get the same total share, which results in each of the three flows to destination host C getting 1/3 of the bandwidth of the three other hosts (which only have one flow each). Similarly, in source fairness mode (third column), the two source hosts share the available capacity, which results in the two flows from source B getting twice the share each compared to the four flows from host A.

In the triple isolation case, we see the flow bandwidths correspond to the quantum scaling outlined in Algorithm 2: The first four flows get their quantum scaled by 1/4 since there are four flows active from host A. The fifth flow gets its quantum scaled by 1/3 since there are three flows active to host C. And finally, the last flow gets its quantum scaled by 1/2 as there are two flows active from host B.

4.2 DiffServ Handling

To demonstrate the DiffServ prioritisation features of CAKE we perform two tests: An RRUL test with each flow marked with a different DiffServ priority, and another test where a high-priority fixed-rate flow competes with several TCP flows.

The result of the former test is seen in Figure 4. This shows that when DiffServ mode is not enabled, all four flows get the same share of the available bandwidth, while in the DiffServ-enabled case, the Best Effort (BE) flow gets most of the bandwidth. This latter effect is important for two reasons: First, it shows that a flow marked as background (BK) is successfully de-prioritised and gets less bandwidth. Secondly, it shows that the high-priority flows (CS5 and EF) are limited so as to not use more than the share of the bandwidth allocated to the high-priority DiffServ classes.

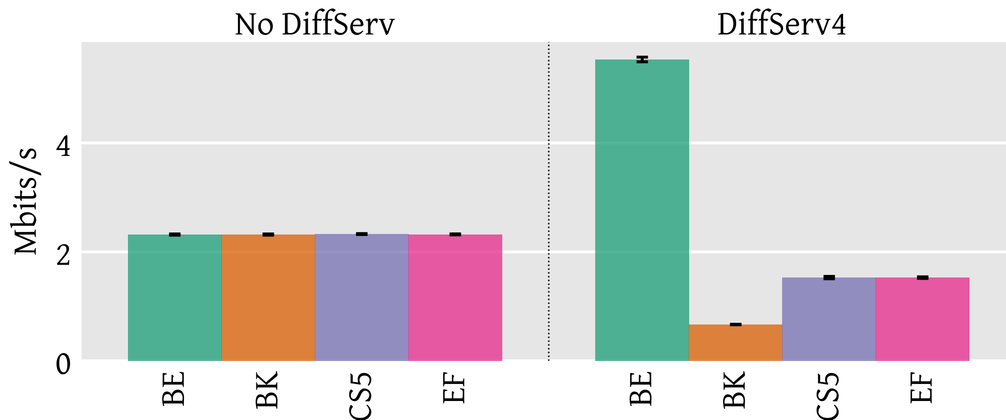


Figure 4: TCP flows on different DiffServ code points.

To look at the latency performance of a high-priority flow, we turn to Figure 5. This shows the latency over time of a fixed-rate 2 Mbps flow, which marks its packets with the high-priority EF DiffServ marking. This is meant to represent a real-time video conversation. In the test, the flow competes with 32 bulk TCP flows. As can be seen in the figure, both FQ-CoDel and CAKE with DiffServ prioritisation disabled fail to ensure low latency for the high-priority flow. Instead, when the bulk flows start after five seconds, a large latency spike is seen, since the real-time flow has to wait for the initial packets of the 32 TCP flows. This causes the real-time flow to build a large queue for itself (since it does not respond to congestion signals), which then drains slowly back to a steady state around 200 ms (for CAKE) or oscillating between 50 and 500 ms (for FQ-CoDel). In contrast, the DiffServ-enabled CAKE keeps the real-time flow completely isolated from the bulk TCP flows, ensuring it sees no added latency over the duration of the test.

4.3 ACK Filtering

Figure 6 shows the performance of ACK filtering on a highly asymmetrical link with 30 Mbps download capacity and only 1 Mbps upload capacity.

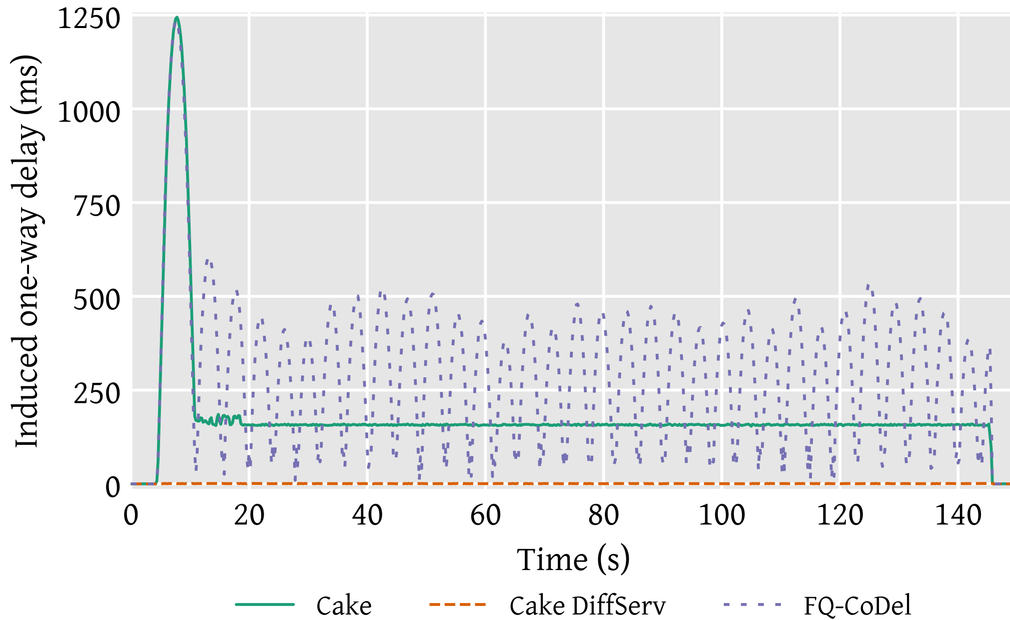


Figure 5: Latency over time of a 2 Mbps fixed-rate flow with 32 competing bulk flows on a 10 Mbps link. The Y-axis shows additional latency above the base latency of 50 ms. The bulk flows start after 5 seconds.

On this link, we run four simultaneous TCP uploads and four simultaneous TCP downloads. The results of this are shown in Figure 6, which shows the aggregate throughput of all four flows in each direction, along with the added latency of a separate measurement flow. Values are normalised to the baseline without ACK filtering to be able to fit on a single graph. As the figure shows, we see a goodput improvement of around 15% in the downstream direction caused by either type of ACK filtering, which shows that insufficient bandwidth for ACKs can impact transfers in the other direction. For upload, the conservative filtering increases goodput by about 10%, while the aggressive filtering increases throughput by as much as 40%, simply by reducing the bandwidth taken up by ACK packets. We attribute the increase in latency to increased congestion in the downlink direction, which is alleviated somewhat by fewer ACKs being queued in the upstream direction in the aggressive case. The absolute magnitude of the latency increase is only 5 ms.

5 Conclusions

CAKE is a comprehensive queue management system for home gateways, that packs several compelling features into an integrated solution, with reasonable defaults to ease configuration. These features include: bandwidth shaping with overhead compensation for various link layers; reasonable DiffServ handling; improved flow hashing with both per-flow and per-host queueing fairness; and filtering of TCP ACKs.

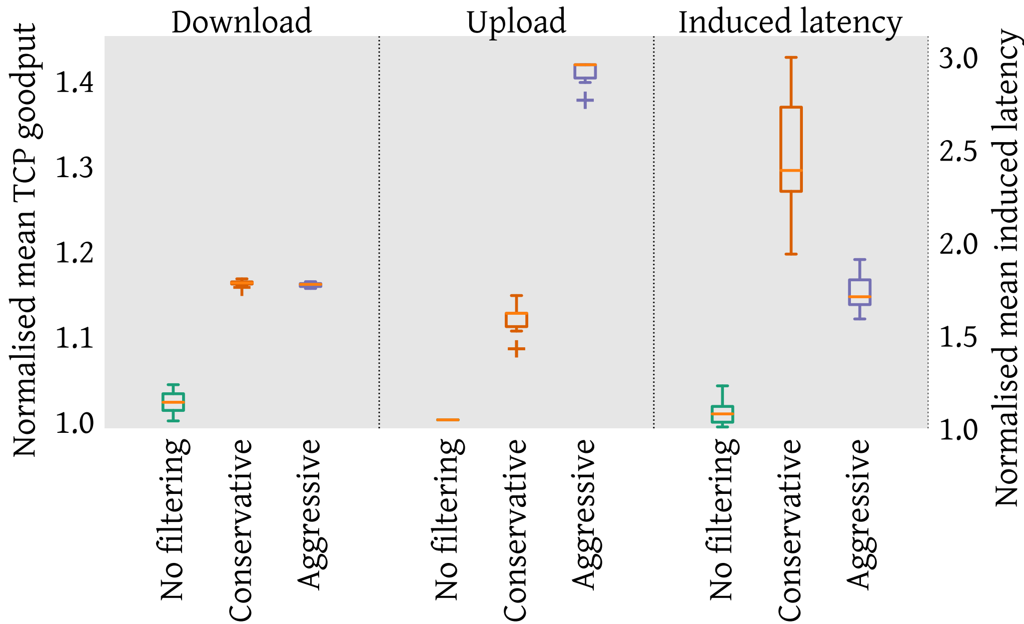


Figure 6: ACK filtering performance on a 30/1 Mbps link. The graph scales are normalised to the "No filtering" case. The download and upload value ranges are 24.5-27.5 Mbps and 0.45-0.7 Mbps, respectively. The latency range is 2.6-7.5 ms.

Our evaluation shows that these features offer compelling advantages, and we believe CAKE has the potential to significantly improve the performance of last-mile internet connections. CAKE is open source and ready for deployment, and already ships in the OpenWrt router firmware distribution.[18]

Acknowledgements

The authors would like to thank the Bufferbloat and OpenWrt communities for their work on the implementation and testing of CAKE. In particular, Kevin Darbyshire-Bryant was instrumental in enabling NAT-awareness, Ryan Mounce contributed the original ACK filtering code, Sebastian Moeller helped get the overhead compensation right and Anil Agarwal helped with the hash collision probability calculations.

References

[1] T. Høiland-Jørgensen, P. Hurtig, and A. Brunstrom, “The Good, the Bad and the WiFi: Modern AQMs in a Residential Setting,” *Computer Networks*, vol. 89, pp. 90 – 106, 2015.

[18]Since the publication of this paper, CAKE has also been accepted into the upstream Linux kernel. It is included starting from kernel version 4.19.

[2] I. Järvinen and M. Kojo, "Evaluating CoDel, PIE, and HRED AQM techniques with load transients," in *39th Annual IEEE Conference on Local Computer Networks*. IEEE, 2014, pp. 159–167.

[3] N. Khademi, D. Ros, and M. Welzl, "The new AQM kids on the block: Much ado about nothing?" Oslo University, Tech. Rep. 434, 2013.

[4] T. Høiland-Jørgensen, D. Täht, and J. Morton, "Piece of CAKE: A Comprehensive Queue Management Solution for Home Gateways," Apr. 2018. https://doi.org/10.5281/zenodo.1226887

[5] J. Corbet, "Network transmit queue limits," LWN Article, August 2011. https://lwn.net/Articles/454390/

[6] T. Høiland-Jørgensen *et al.*, "Ending the Anomaly: Achieving Low Latency and Airtime Fairness in WiFi," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. Santa Clara, CA: USENIX Association, 2017, pp. 139–151.

[7] G. Niestegge, "The 'leaky bucket' policing method in the ATM (Asynchronous Transfer Mode) network," *International Journal of Communication Systems*, vol. 3, no. 2, pp. 187–197, 1990.

[8] A. Eleftheriadis and D. Anastassiou, "Constrained and general dynamic rate shaping of compressed digital video," in *International Conference on Image Processing*, vol. 3. IEEE, 1995, pp. 396–399.

[9] V. P. Rao, M. P. Tahiliani, and U. K. K. Shenoy, "Analysis of sfqCoDel for active queue management," in *ICADIWT 2014*. IEEE, 2014.

[10] R. Adams, "Active Queue Management: A survey," *IEEE Communications Surveys & Tutorials*, vol. 15, no. 3, pp. 1425–1476, 2013.

[11] N. Benameur, F. Guillemin, and L. Muscariello, "Latency Reduction in Home Access Gateways with Shortest Queue First," in *ISOC Workshop on Reducing Internet Latency*, 2013.

[12] T. Høiland-Jørgensen *et al.*, "The Flow Queue CoDel Packet Scheduler and Active Queue Management Algorithm," RFC 8290, RFC Editor, Jan. 2018.

[13] M. Shreedhar and G. Varghese, "Efficient fair queuing using deficit round-robin," *IEEE/ACM Transactions on Networking*, Jun. 1996.

[14] A. Kortebi, S. Oueslati, and J. Roberts, "Implicit service differentiation using deficit round robin," *19th International TELETRAFFIC CONGRESS*, 2005.

[15] A. Kortebi, S. Oueslati, and J. W. Roberts, "Cross-Protect: Implicit Service Differentiation and Admission Control," in *2004 Workshop on High Performance Switching and Routing*. IEEE, 2004, pp. 56 – 60.

[16] J. Babiarz, K. Chan, and F. Baker, "Configuration Guidelines for DiffServ Service Classes," RFC 4594 (Informational), RFC Editor, Aug. 2006.

[17] T. Szigeti, J. Henry, and F. Baker, "Mapping Diffserv to IEEE 802.11," RFC 8325 (Proposed Standard), RFC Editor, Feb. 2018.

[18] W.-S. Hwang and P.-C. Tseng, "A QoS-aware residential gateway with bandwidth management," *IEEE Transactions on Consumer Electronics*, vol. 51, no. 3, pp. 840–848, 2005.

[19] H. Wu *et al.*, "ACK filtering on bandwidth asymmetry networks," in *Proceedings of the Fifth Asia-Pacific Conference on Communications and Fourth Optoelectronic and Communications Conference*. IEEE, 1999.

[20] L. Storfer, "Enhancing cable modem TCP performance," *Texas Instruments Inc. white paper*, 2003.

[21] H. Kim *et al.*, "On the cross-layer impact of TCP ACK thinning on IEEE 802.11 wireless MAC dynamics," in *64th Vehicular Technology Conference*. IEEE, 2006.

[22] A. J. Smith, "A comparative study of set associative memory mapping algorithms and their use for cache and main memory," *IEEE Transactions on Software Engineering*, no. 2, pp. 121–130, 1978.

[23] T. Høiland-Jørgensen *et al.*, "Flent: The FLExible Network Tester," in *ValueTools 2017*, 2017.

[24] D. Taht, "Realtime response under load (rrul) test," November 2012. https://www.bufferbloat.net/projects/bloat/wiki/RRUL_Spec/

Paper V

Ending the Anomaly

Achieving Low Latency and Airtime Fairness in WiFi

Reprinted from

2017 USENIX Annual Technical Conference
(USENIX ATC 17), July 12–14 2017, Santa Clara, CA

“There’s nothing special about wireless networks except that wireless capacity is sometimes less than what you can get, for example, from optical fiber.”

Vint Cerf

Ending the Anomaly

Achieving Low Latency and Airtime Fairness in WiFi

Toke Høiland-Jørgensen, Michal Kazior, Dave Täht, Per Hurtig and Anna Brunstrom

toke.hoiland-jorgensen@kau.se, michal.kazior@tieto.com, dave@taht.net, per.hurtig@kau.se, anna.brunstrom@kau.se

Abstract

With more devices connected, delays and jitter at the WiFi hop become more prevalent, and correct functioning during network congestion becomes more important. However, two important performance issues prevent modern WiFi from reaching its potential: increased latency under load caused by excessive queueing (i.e. *bufferbloat*) and the 802.11 performance anomaly.

To remedy these issues, we present a novel two-part solution. We design a new queueing scheme that eliminates bufferbloat in the wireless setting. Leveraging this queueing scheme, we then design an airtime fairness scheduler that operates at the access point and doesn't require any changes to clients.

We evaluate our solution using both a theoretical model and experiments in a testbed environment, formulating a suitable analytical model in the process. We show that our solution achieves an order of magnitude reduction in latency under load, large improvements in multi-station throughput, and nearly perfect airtime fairness for both TCP and downstream UDP traffic. Further experiments with application traffic confirm that the solution provides significant performance gains for real-world traffic.We develop a production quality implementation of our solution in the Linux kernel, the platform powering most access points outside of the managed enterprise setting. The implementation has been accepted into the mainline kernel distribution, making it available for deployment on billions of devices running Linux today.

1 Introduction

As more mobile devices connect to the internet, and internet connections increase in capacity, WiFi is increasingly the bottleneck for users of the internet.

This means that congestion at the WiFi hop becomes more common, which in turn increases the potential for bufferbloat at the WiFi link, severely degrading performance [1].

The 802.11 performance anomaly [2] also negatively affects the performance of WiFi bottleneck links. This is a well-known property of WiFi networks: if devices on the network operate at different rates, the MAC protocol will ensure *throughput fairness* between them, meaning that all stations will effectively transmit at the lowest rate. The anomaly was first described in 2003, and several mitigation strategies have been proposed in the literature (e.g., [3, 4]), so one would expect the problem to be solved. However, none of the proposed solutions have seen widespread real-world deployment.

Recognising that the solutions to these two problems are complementary, we design a novel queue management scheme that innovates upon previous solutions to the bufferbloat problem by adapting it to support the 802.11 suite of WiFi protocols. With this queueing structure in place, eliminating the performance anomaly becomes possible by scheduling the queues appropriately. We develop a deficit-based airtime fairness scheduler to achieve this.

We implement our solution in the WiFi stack of the Linux kernel. Linux is perhaps the most widespread platform for commercial off-the-shelf routers and access points outside the managed enterprise, and hundreds of millions of users connect to the internet through a Linux-based gateway or access point on a daily basis. Thus, while our solution is generally applicable to any platform that needs to support WiFi, using Linux as our example platform makes it possible to validate that our solution is of production quality, and in addition gives valuable insights into the practical difficulties of implementing these concepts in a real system.

The rest of this paper describes our solution in detail, and is structured as follows: Section 2 describes the bufferbloat problem in the context of WiFi and the WiFi performance anomaly, and shows the potential performance improvement from resolving them. Section 3 describes our proposed solution in detail and Section 4 presents our experimental evaluation. Finally, Section 5 summarises related work and Section 6 concludes.

2 Background

In this section we describe the two performance issues we are trying to solve – Bufferbloat in the WiFi stack and the 802.11 performance anomaly. We explain why these matter, and show the potential benefits from solving them.

2.1 Bufferbloat in the context of WiFi

Previous work on eliminating bufferbloat has shown that the default buffer sizing in many devices causes large delays and degrades performance. It also shows that this can be rectified by introducing modern queue management to the bottleneck link [1, 5, 6]. However, this does not work as well for WiFi; prior work has shown that neither decreasing buffer sizes [7] nor applying

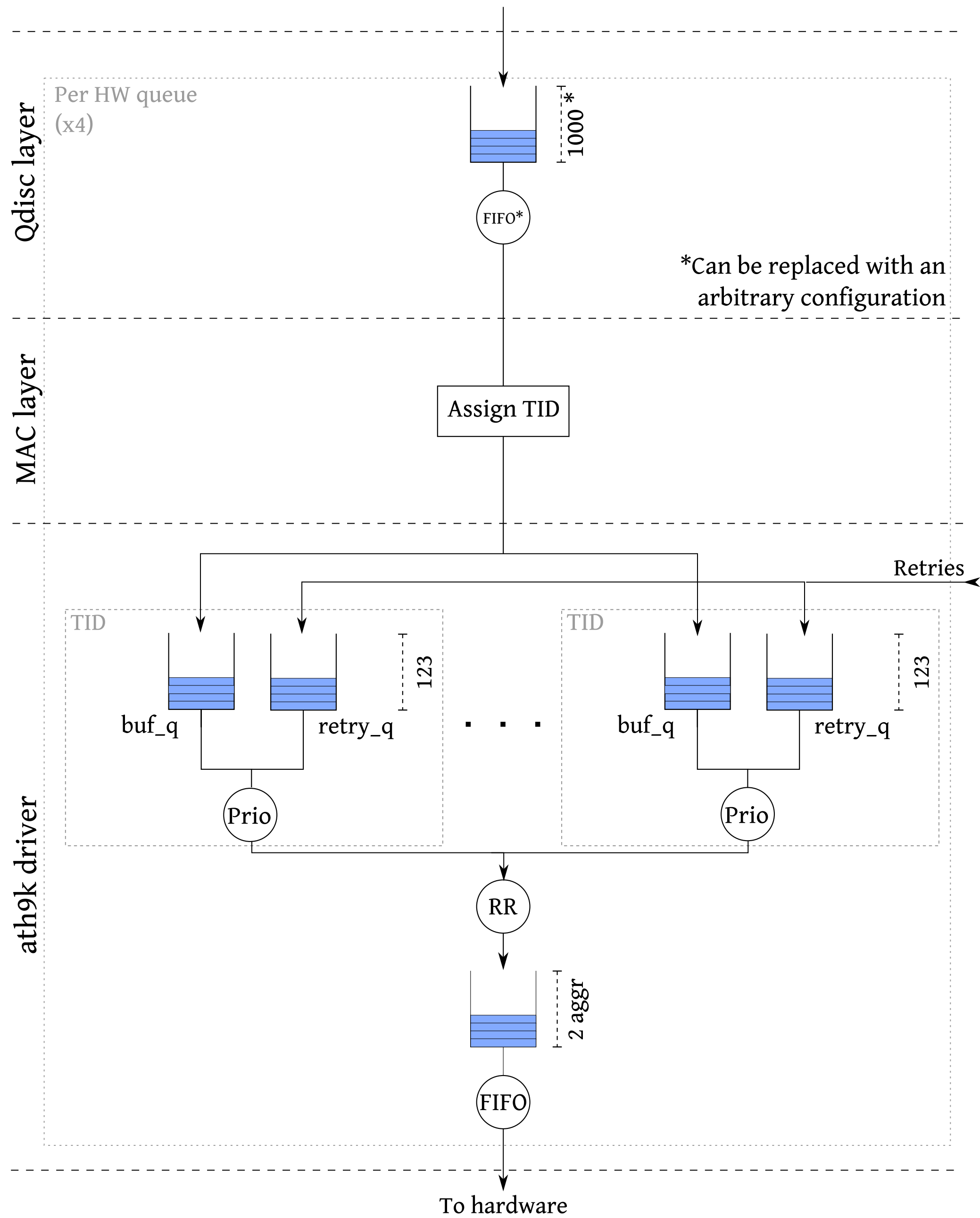


Figure 1: The queueing structure of the Linux WiFi stack.

queue management algorithms to the WiFi interface [1] can provide the same reduction in latency under load as for wired links.

The reason for the limited effect of prior solutions is queueing in the lower layers of the wireless network stack. For Linux, this is clearly seen in the queueing structure, depicted in Figure 1. The upper queue discipline ("qdisc") layer, which is where the advanced queue management schemes can be installed, sits above both the mac80211 subsystem (which implements the base 802.11 protocol) and the driver. As the diagram shows, there is significant unmanaged queueing in these lower layers, limiting the efficacy of the queue management schemes and leading to increased delay. Such a design is typical for an environment where low-level protocol details impose a certain queueing structure (as opposed to a wired Ethernet network, where the protocol-specific processing performed by the driver does not necessitate queueing). In WiFi this queueing is needed to build aggregates (and to a lesser extent to keep the hardware busy within the time constrains imposed by the protocol), but a similar situation can be seen in, e.g., mobile broadband devices, DSL modem drivers, and even in some VPN protocols, where the encryption processing can require a separate layer of queueing.

To solve this, an integrated queueing scheme is needed, that applies modern queue management to the protocol-specific queueing structures. In Section 3 we describe our design of such a solution for the WiFi domain. Figure 2 showcases the gain from applying our solution. The figure shows a latency measurement (ICMP ping) performed simultaneously with a simple TCP download to each of the stations on the network. The dashed line shows the state of the Linux kernel before we applied our solution, with several hundred milliseconds of added latency. The solid line shows the effects of applying the solution we propose in this paper – a latency reduction of an order of magnitude.

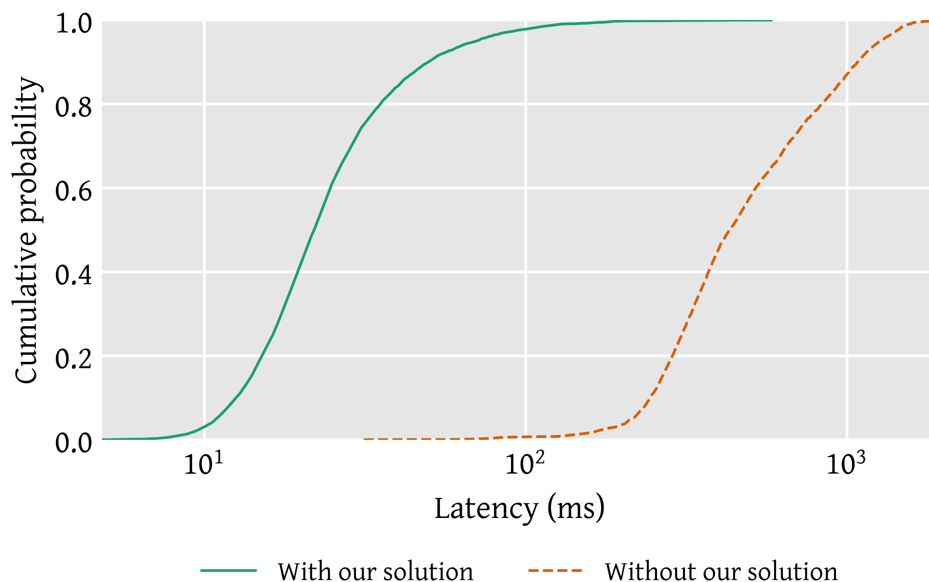


Figure 2: Latency of an ICMP ping flow with simultaneous TCP download traffic, before and after our modifications.

2.2 Airtime fairness

The 802.11 performance anomaly was first described for the 802.11b standard in [2], which showed that in a wireless network with differing rates, each station would achieve the same effective throughput even when their rates were different. Later work has shown both analytically and experimentally that time-based fairness improves the aggregate performance of the network [3], and that the traditional notion of proportional fairness [8] translates to airtime fairness when applied to a WiFi network [9].

This latter point is an important part of why airtime fairness is desirable – proportional fairness strikes a balance between network efficiency and allowing all users a minimal level of service. Since a wireless network operates over a shared medium (the airwaves), access to this medium is the scarce resource that needs to be regulated. Achieving airtime fairness also has the desirable property that it makes a station's performance dependent on the *number* of active stations in the network, and not on the performance of each of those other stations.

The addition of packet aggregation to WiFi (introduced in 802.11n and also present in 802.11ac) adds some complexity to the picture. To quantify the expected gains of airtime fairness in the context of these newer revisions of 802.11, the following section develops an analytical model to predict throughput and airtime usage.

2.2.1 An analytical model for 802.11 with aggregation

The models in [2] and [3] give analytical expressions for expected throughput and airtime share for 802.11b (the latter also under the assumption of airtime fairness). Later work [10] updates this by developing analytical expressions for packet sizes and transmission times for a single station using 802.11n. However, this work does not provide expressions for predicting throughput and airtime usage. In this section we expand on the work of [10] to provide such an expression. While we focus on 802.11n here, the 802.11ac standard is backwards-compatible with 802.11n as far as the aggregation format is concerned, so these calculations apply to the newer standard as well.

For the following exposition, we assume a set of active stations, I. Each station, i, transmits aggregates of a fixed size of L_i bytes. In practice, the aggregates are composed of data packets, plus overhead and padding. The 802.11n standard permits two types of aggregation (known as A-MPDU and A-MSDU), which differ in how they combine packets into MAC-layer aggregates. For A-MPDU aggregation (which is the most common in use in 802.11n devices), the size of an aggregate consisting of n_i packets of size l_i is given by:

$$L_i = n_i(l_i + L_{delim} + L_{mac} + L_{FCS} + L_{pad}) \qquad (1)$$

where $L_{delim}, L_{mac}, L_{FCS}, L_{pad}$ are, respectively, the frame delimiter, MAC header, frame check sequence and frame padding. However, these details are not strictly necessary for our exposition, so we leave them out in the

Table 1: Calculated airtime, calculated rate and measured rate for the three stations (two fast and one slow) in our experimental setup. The aggregation size is the measured mean aggregation size (in bytes) from our experiments and the measured rates (Exp column) are mean UDP throughput values.

Aggr size	$T(i)$	Rates (Mbps)			
		PHY	Base	$R(i)$	Exp
Baseline (FIFO queue)[1]					
6892	10%	144.4	97.3	9.7	7.1
7833	11%	144.4	101.1	11.4	6.3
2914	79%	7.2	6.5	5.1	5.3
Total				26.4	18.7
Airtime Fairness					
28434	33%	144.4	126.7	42.2	38.8
28557	33%	144.4	126.8	42.3	35.6
2914	33%	7.2	6.5	2.2	2.0
Total				86.8	76.4

following and instead refer to [10] for a nice overview of the details of aggregate composition.

A station transmits data over the air at a particular data rate r_i (measured in bits per second). So the time to transmit the data portion of an aggregate is simply:

$$T_{data}(i) = \frac{8L_i}{r_i} \tag{2}$$

From this we can compute the expected effective station rate, assuming no errors or collisions, and no other active stations:

$$R_0(i) = \frac{L_i}{T_{data}(i) + T_{oh}} \tag{3}$$

where T_{oh} is the per-transmission overhead, which consists of the frame header, the inter-frame spacing, the average block acknowledgement time, and the average back-off time before transmission. We again leave out the details and point interested readers to [10, 11].

Turning to airtime fairness, we borrow two insights from the analysis in [3]:

1. The rate achieved by station i is simply given by the baseline rate it can achieve when no other stations are present (i.e., $R_0(i)$) multiplied by the share of airtime available to the station.

[1]The aggregation size and throughput values vary quite a bit for this test, because of the randomness of the FIFO queue emptying and filling. We use the median value over all repetitions of the per-test mean throughput and aggregation size; see the online appendix for graphs with error bars.

2. When airtime fairness is enforced, the airtime is divided equally among the stations (by assumption). When it is not, the airtime share of station i is the ratio between the time that station spends on a single transmission (i.e., $T_{data}(i)$) and the total time all stations spend doing one transmission each.

With these points in mind, we express the expected airtime share $T(i)$ and rate $R(i)$ as:

$$T(i) = \begin{cases} \frac{1}{|I|} & \text{with fairness} \\ \frac{T_{data}(i)}{\sum_{j \in I} T_{data}(j)} & \text{otherwise} \end{cases} \quad (4)$$

$$R(i) = T(i) R_0(i) \quad (5)$$

Using the above, we can calculate the expected airtime share and effective rate for each station in our experimental setup. The assumption of no contention holds because all data is transmitted from the access point. As the queueing structure affects the achievable aggregation level (and thus the predictions of the model), we use the measured average aggregation levels in our experiments as input to the model.

The model predictions, along with the actual measured throughput in our experiments, are shown in Table 1. The values will be discussed in more detail in Section 4, so for now we will just remark that this clearly shows the potential of eliminating the performance anomaly: An increase in total throughput by up to a factor of five.

3 Our solution

We focus on the access point scenario in formulating our solution, since a solution that only requires modifying the access point makes deployment easier as there are fewer devices to upgrade. However, WiFi client devices can also benefit from the proposed queueing structure. And while we have focused on 802.11n here, the principles apply equally to both earlier (802.11abg) and newer (802.11ac) standards. The rest of this section describes the two parts of our solution, and outlines the current implementation status in Linux.

3.1 A bloat-free queueing structure for 802.11

An operating system networking stack has many layers of intermediate queueing between different subsystems, each of which can add latency. For specialised systems, it is possible to remove those queues entirely, which achieves significant latency reductions [12]. While such a radical restructuring of the operating system is not always possible, the general principle of collapsing multiple layers of queues can be applied to the problem of reducing bufferbloat in WiFi.

As mentioned in Section 2.1, an integrated queueing structure is needed to deal with protocol-specific constraints while still eliminating bufferbloat.

What we propose here is such an integrated structure that is specifically suited to the 802.11 MAC. The components we use to build this structure already exists in various forms; the novelty of our solution lies in their integration, and some algorithmic innovations to make the implementation feasible, even on small devices.

There are three main constraints we must take into account when designing our queueing scheme. First, we must be able to handle aggregation; the 802.11e standard specifies that packets can be assigned different Traffic Identifiers (TIDs) (typically based on their DiffServ markings [13]), and the 802.11n standard specifies that aggregation be performed on a per-TID basis. Second, we must have enough data processed and ready to go when the hardware wins a transmit opportunity; there is not enough time to do a lot of processing at that time. Third, we must be able to handle packets that are re-injected from the hardware after a failed transmission; these must be re-transmitted ahead of other queued packets, as transmission can otherwise stall due to a full Block Acknowledgement Window.

The need to support aggregation, in particular, has influenced our proposed design. A generic packet queueing mechanism, such as that in the Linux qdisc layer (see Section 2.1), does not have the protocol-specific knowledge to support the splitting of packets into separate queues, as is required for aggregation. And introducing an API to communicate this knowledge to the qdisc layer would impose a large complexity cost on this layer, to the detriment of network interfaces that do not have the protocol-specific requirements. So rather than modifying the generic queueing layer, we bypass it completely, and instead incorporate the smart queue management directly into the 802.11 protocol-specific subsystem. The main drawback of doing this is, of course, a loss of flexibility. With this design, there is no longer a way to turn off the smart queue management completely; and it does add some overhead to the packet processing. However, as we will see in the evaluation section, the benefits by far outweigh the costs.

We build our smart queue management solution on the FQ-CoDel queue management scheme, which has been shown to be a best-in-class bufferbloat mitigation technique [1, 5, 6]. The original FQ-Codel algorithm is a hybrid fairness queueing and AQM algorithm [14]. It functions as a Deficit Round-Robin (DRR) scheduler [15] between flows, hashing packets into queues based on their transport protocol flows, and applying the CoDel AQM separately to each queue, in order to keep the latency experienced by each flow under control. FQ-CoDel also adds an optimisation for sparse flows to the basic DRR algorithm. This optimisation allows flows that use less than their fair share of traffic to gain scheduling priority, reducing the time their packets spend in the queue. For a full explanation of FQ-CoDel, see [14].

FQ-CoDel allocates a number of sub-queues that are used for per-flow scheduling, and so simply assigning a full instance of FQ-CoDel to each TID is impractical. Instead, we innovate on the FQ-CoDel design by having it operate on a fixed total number of queues, and group queues based on which TID they are associated with. So when a packet is hashed and assigned to a queue, that

Algorithm 3 802.11 queue management algorithm - enqueue.

```
 1: function ENQUEUE(pkt, tid)
 2:     if queue_limit_reached() then                          ▷ Global limit
 3:         drop_queue ← FIND_LONGEST_QUEUE()
 4:         DROP(drop_queue.head_pkt)
 5:     queue ← HASH(pkt)
 6:     if queue.tid ≠ NULL and queue.tid ≠ tid then
 7:         queue ← tid.overflow_queue                         ▷ Hash collision
 8:     queue.tid ← tid
 9:     TIMESTAMP(pkt)                                ▷ Used by CoDel at dequeue
10:     APPEND(pkt, queue)
11:     if queue is not active then
12:         LIST_ADD(queue, tid.new_queues)
```

queue is in turn assigned to the TID the packet is destined for. In case that queue is already active and assigned to another TID (which means that a hash collision has occurred), the packet is instead queued to a TID-specific overflow queue.[20] A global queue size limit is kept, and when this is exceeded, packets are dropped from the globally longest queue, which prevents a single flow from locking out other flows on overload. The full enqueue logic is shown in Algorithm 3.

The lists of active queues are kept in a per-TID structure, and when a TID needs to dequeue a packet, the FQ-CoDel scheduler is applied to the TID-specific lists of active queues. This is shown in Algorithm 4.

The obvious way to handle the two other constraints mentioned above (keeping the hardware busy, and handling retries), is, respectively, to add a small queue of pre-processed aggregates, and to add a separate priority queue for packets that need to be retried. And indeed, this is how the ath9k driver already handled these issues, so we simply keep this. The resulting queueing structure is depicted in Figure 3.

3.2 Airtime fairness scheduling

Given the above queueing structure, achieving airtime fairness becomes a matter of measuring the airtime used by each station, and appropriately scheduling the order in which stations are served. For each packet sent or received, the packet duration can either be extracted directly from a hardware register, or it can be calculated from the packet length and the rate at which it was sent (including any retries). Each packet's duration is subtracted from a per-station airtime deficit which is used by a deficit scheduler, modelled after FQ-CoDel, to decide the destination station ahead of each transmission. The decision to keep the deficit per station instead of per TID follows from the fact

[20] A hash collision can of course also mean that two flows assigned to the same TID end up in the same queue. In this case, no special handling is needed, and the two flows will simply share a queue like in any other hash-based fairness queueing scheme.

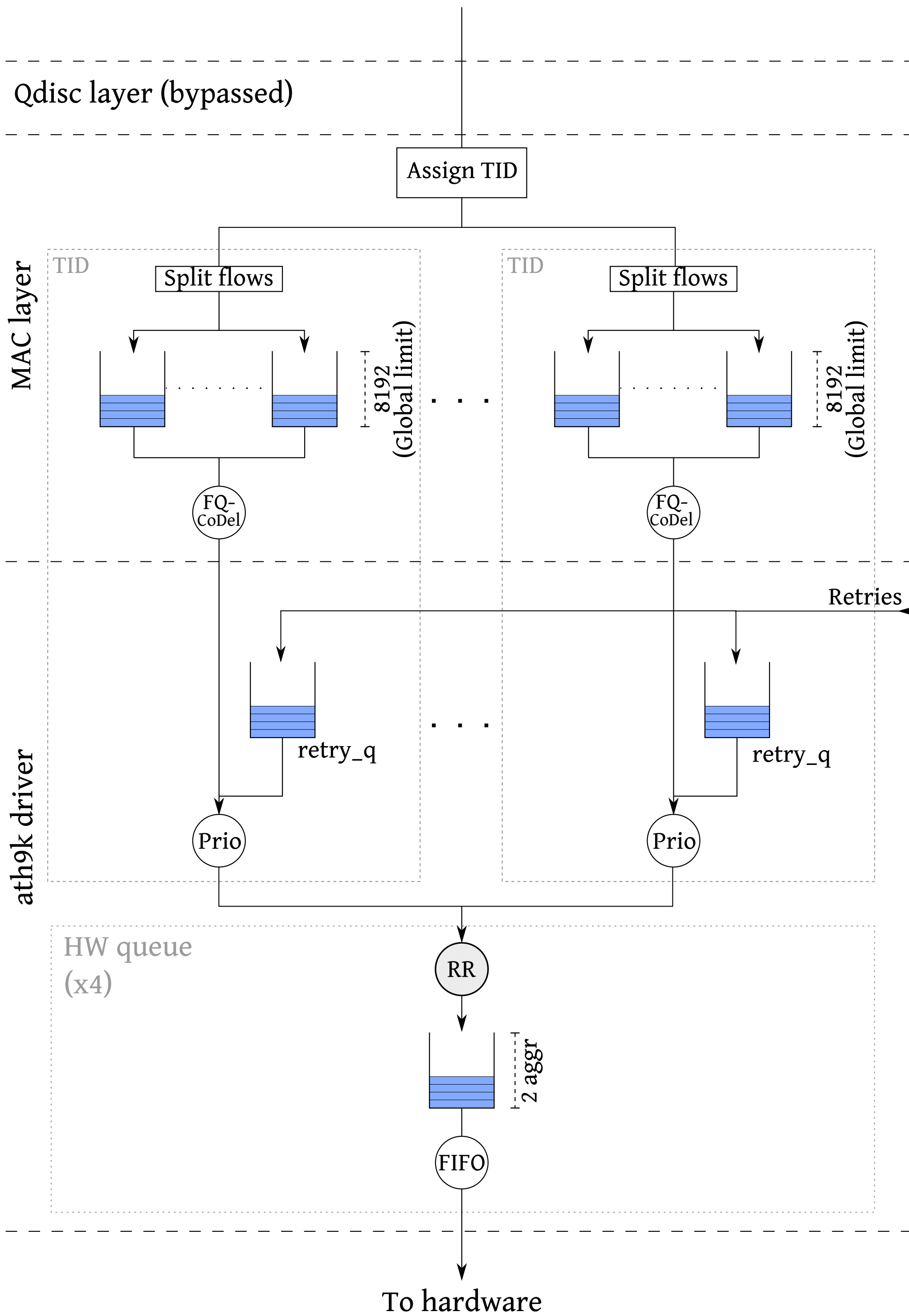


Figure 3: Our 802.11-specific queueing structure, as it looks when applied to the Linux WiFi stack.

Algorithm 4 802.11 queue management algorithm - dequeue.

```
 1: function DEQUEUE(tid)
 2:     if tid.new_queues is non-empty then
 3:         queue ← LIST_FIRST(tid.new_queues)
 4:     else if tid.old_queues is non-empty then
 5:         queue ← LIST_FIRST(tid.old_queues)
 6:     else
 7:         return NULL
 8:     if queue.deficit ≤ 0 then
 9:         queue.deficit ← queue.deficit + quantum
10:         LIST_MOVE(queue, tid.old_queues)
11:         restart
12:     pkt ← CODEL_DEQUEUE(queue)
13:     if pkt is NULL then                                  ▷ queue empty
14:         if queue ∈ tid.new_queues then
15:             LIST_MOVE(queue, tid.old_queues)
16:         else
17:             LIST_DEL(queue)
18:             queue.tid ← NULL
19:         restart
20:     queue.deficit ← queue.deficit − pkt.length
21:     return pkt
```

that the goal of airtime fairness is to even out differences in the physical signal conditions, which is a per-station property. However, because the four 802.11 QoS precedence markings (VO, VI, BE and BK) are commonly scheduled independently down to the hardware level, we actually keep four deficits per station, corresponding to the four precedence levels, to simplify the scheduler implementation.

The resulting airtime fairness scheduler is shown in Algorithm 5. It is similar to the the FQ-CoDel dequeue algorithm, with stations taking the place of flows, and the deficit being accounted in microseconds instead of bytes. The two main differences are (1) that the scheduler function loops until the hardware queue becomes full (at two queued aggregates), rather than just dequeueing a single packet; and (2) that when a station is chosen to be scheduled, it gets to build a full aggregate rather than a single packet.

Compared to the closest previously proposed solution [16], our scheme has several advantages:

1. We lower implementation complexity by leveraging existing information on per-aggregate transmission rates and time, and by using a per-station deficit instead of token buckets, which means that no token bucket accounting needs to be performed at TX and RX completion time.

2. [16] measures time from an aggregate is submitted to the hardware until it is sent, which risks including time spent waiting for other stations to transmit. We increase accuracy by measuring the actual time spent

transmitting, and by also accounting the airtime from received frames to each station's deficit.

3. We improve on the basic scheduler design by adding an optimisation for sparse stations, analogous to FQ-CoDel's sparse flow optimisation. This improves latency for stations that only transmit occasionally, by giving them temporary priority for one round of scheduling. We apply the same protection against gaming this mechanism that FQ-CoDel does to its sparse flow mechanism [14].

Algorithm 5 Airtime fairness scheduler. The schedule function is called on packet arrival and on transmission completion.

```
1:  function SCHEDULE
2:      while hardware queue is not full do
3:          if new_stations is non-empty then
4:              station ← LIST_FIRST(new_stations)
5:          else if old_stations is non-empty then
6:              station ← LIST_FIRST(old_stations)
7:          else
8:              return
9:          deficit ← station.deficit[pkt.qoslvl]
10:         if deficit ≤ 0 then
11:             station.deficit[pkt.qoslvl] ← deficit + quantum
12:             LIST_MOVE(station, old_stations)
13:             restart
14:         if station's queue is empty then
15:             if station ∈ new_stations then
16:                 LIST_MOVE(station, old_stations)
17:             else
18:                 LIST_DEL(station)
19:             restart
20:         BUILD_AGGREGATE(station)
```

3.3 Implementation

We have implemented our proposed queueing scheme in the Linux kernel, modifying the mac80211 subsystem to include the queueing structure itself, and modifying the ath9k and ath10k drivers for Qualcomm Atheros 802.11n and 802.11ac chipsets to use the new queueing structure. The airtime fairness scheduler implementation is limited to the ath9k driver, as the ath10k driver lacks the required scheduling hooks.

Our modifications have been accepted into the mainline Linux kernel, different parts going into kernel releases 4.8 through 4.11, and is included in the LEDE open source router firmware from release 17.01. The implementation

is available online, as well as details about our test environment and the full evaluation dataset.[21]

4 Evaluation

We evaluate our modifications in a testbed setup consisting of five PCs: Three wireless clients, an access point, and a server located one Gigabit Ethernet hop from the access point, which serves as source and sink for the test flows. All the wireless nodes are regular x86 PCs equipped with PCI-Express Qualcomm Atheros AR9580 adapters (which use the ath9k driver). Two of the test clients are placed in close proximity to the access point (and are referred to as fast nodes), while the last (referred to as the slow node) is placed further away and configured to only support the MCS0 rate, giving a maximum throughput to that station of 7.2 Mbps at the PHY layer. A fourth virtual station is added as an additional fast node to evaluate the sparse station optimisation (see Section 4.1.4 below). All tests are run in HT20 mode on an otherwise unused channel in the 5Ghz band. We use 30 test repetitions of 30 seconds each unless noted otherwise.

The wireless nodes run an unmodified Ubuntu 16.04 distribution. The access point has had its kernel replaced with a version 4.6 kernel from kernel.org on top of which we apply our modifications. We run all experiments with four queue management schemes, as follows:

FIFO: The default 4.6 kernel from kernel.org modified only to collect the airtime used by stations, running with the default PFIFO queueing discipline installed on the wireless interface.

FQ-CoDel: As above, but using the FQ-CoDel qdisc on the wireless interface.

FQ-MAC: Kernel patched to include the FQ-CoDel based intermediate queues in the MAC layer (patching the mac80211 subsystem and the ath9k driver).

Airtime fair FQ: As FQ-MAC, but additionally including our airtime fairness scheduler in the ath9k driver.

Our evaluation is split into two parts. First, we validate the effects of the modifications in simple scenarios using synthetic benchmark traffic. Second, we evaluate the effect of our modifications on two application traffic scenarios, to verify that they provide a real-world benefit.

4.1 Validation of effects

In this section we present the evaluation of our modifications in simple synthetic scenarios designed to validate the correct functioning of the algorithms and to demonstrate various aspects of their performance.

[21] See `https://www.cs.kau.se/tohojo/airtime-fairness/` for the online appendix that contains additional material, as well as the full experimental dataset and links to the relevant Linux code.

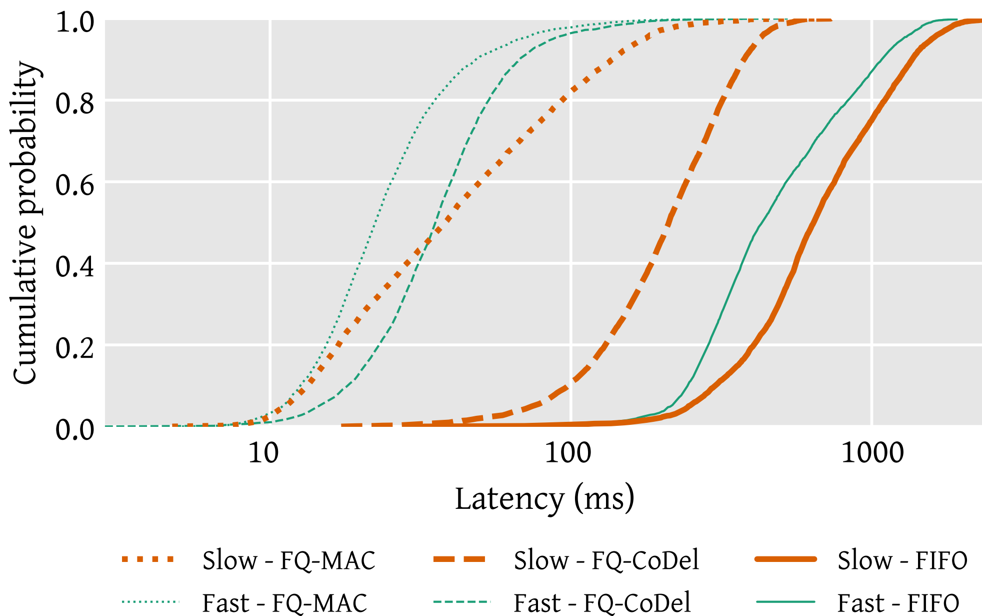


Figure 4: Latency (ICMP ping) with simultaneous TCP download traffic.

4.1.1 Latency reductions

Figure 4 is the full set of results for our ICMP latency measurements with simultaneous TCP download traffic (of which a subset was shown earlier in Figure 2). Here, the FIFO case shows several hundred milliseconds of latency when the link is saturated by a TCP download. FQ-CoDel alleviates this somewhat, but the slow station still sees latencies of more than 200 ms in the median, and the fast stations around 35 ms. With the FQ-MAC queue restructuring, this is reduced so that the slow station now has the same median latency as the fast one does in the FQ-CoDel case, while the fast stations get their latency reduced by another 45%. The airtime scheduler doesn't improve further upon this, other than to alter the shape of the distribution slightly for the slow station (but retaining the same median). For this reason, we have omitted it from the figure to make it more readable.

For simultaneous upload and download the effect is similar, except that in this case the airtime scheduler slightly worsens the latency to the slow station, because it is scheduled less often to compensate for its increased airtime usage in the upstream direction. The graph of this case can be found in the online appendix.

4.1.2 Airtime usage

Figure 5 shows the airtime usage of the three active stations for one-way UDP traffic going to the stations. There is no reverse traffic and no contention between stations, since only the access point is transmitting data. This is the

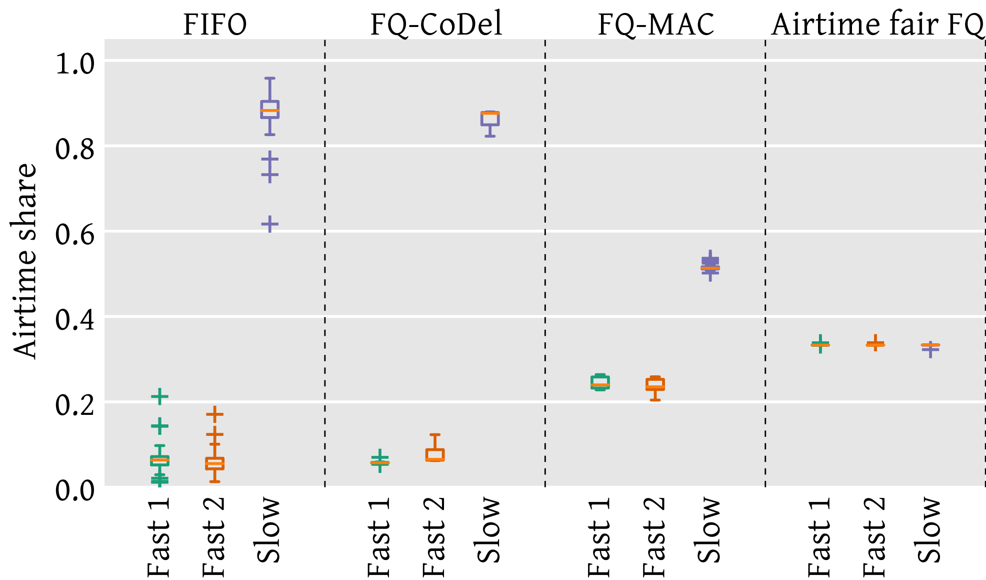


Figure 5: Airtime usage for one-way UDP traffic. Each column shows the relative airtime usage of one of the three stations, with the four sections corresponding to the four queue management schemes.

simplest case to reason about and measure, and it clearly shows the performance anomaly is present in the current Linux kernel (left half of the figure): The third station (which transmits at the lowest rate) takes up around 80% of the available airtime, while the two other stations share the remaining 20%.

The differences between the first two columns and the third column are due to changes in aggregation caused by the change to the queueing structure. In the FIFO and FQ-CoDel cases, there is a single FIFO queue with no mechanism to ensure fair sharing of that queue space between stations. So because the slow station has a lower egress rate, it will build more queue until it takes up the entire queueing space. This means that there are not enough packets queued to build sufficiently large aggregates for the fast stations to use the airtime effectively. The FQ-MAC queueing scheme drops packets from the largest queue on overflow, which ensures that the available queueing space is shared between stations, which improves aggregation for the fast stations and thus changes the airtime shares. Referring back to Table 1, the values correspond well to those predicted by the analytical model. The fourth column shows the airtime fairness scheduler operating correctly – each station receives exactly the same amount of airtime in this simple one-way test.

Going beyond the simple UDP case, Figure 6 shows Jain's fairness index for the airtime of the four different schemes for UDP (for comparison) and both unidirectional (to the clients) and bidirectional (simultaneous up and down) TCP traffic. The same general pattern is seen with TCP as with UDP traffic: The performance anomaly is clear for the FIFO case, but somewhat lessened for the FQ-CoDel and FQ-MAC cases. The airtime fairness scheduler achieves close to perfect sharing of airtime in the case of uni-directional traffic, with a

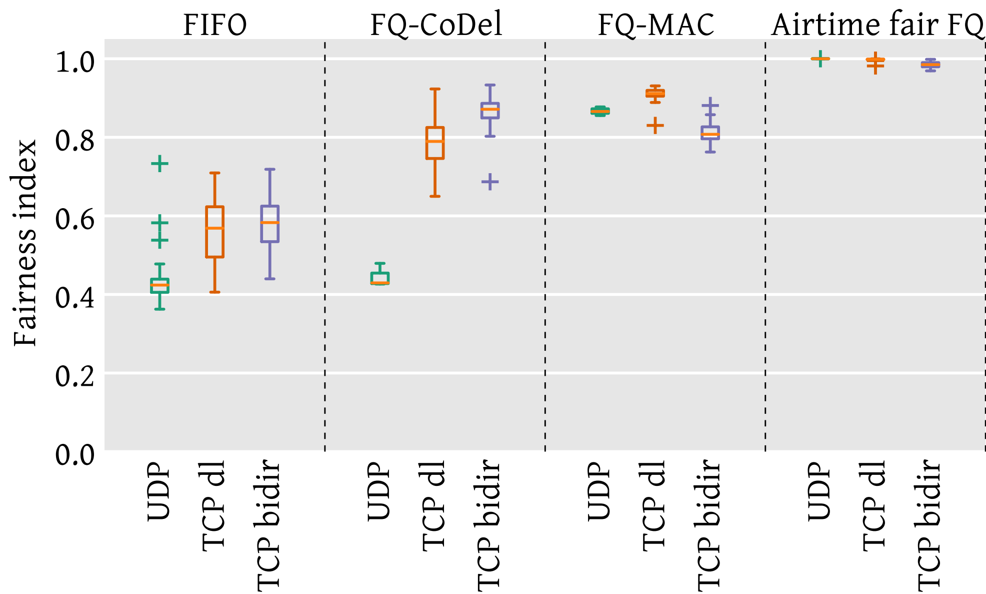


Figure 6: Jain's fairness index (computed over the airtime usage of the three stations) for UDP traffic, TCP download, and simultaneous TCP upload and download traffic.

slight dip for bidirectional traffic. The latter is because the scheduler only exerts indirect control over the traffic sent from the clients, and so cannot enforce perfect fairness as with the other traffic types. However, because airtime is also accounted for received packets, the scheduler can partially compensate, which is why the difference between the unidirectional and bidirectional cases is not larger than it is.

4.1.3 Effects on throughput

As was already shown in Table 1, fixing the performance anomaly improves the efficiency of the network for unidirectional UDP traffic. Figure 7 shows the throughput for downstream TCP traffic. For this case, the fast stations increase their throughput as fairness goes up, and the slow station decreases its throughput. The total effect is a net increase in throughput. The increase from the FIFO case to FQ-CoDel and FQ-MAC is due to better aggregation for the fast stations. This was observed for UDP as well in the case of FQ-MAC, but for FQ-CoDel the slow station would occupy all the queue space in the driver, preventing the fast station from achieving full aggregation. With the TCP feedback loop in place, this lock-out behaviour is lessened, and so fast stations increase their throughput.

When traffic is flowing in both directions simultaneously, the pattern is similar, but with a slightly higher variance. The graph for the bidirectional case can be found in the online appendix.

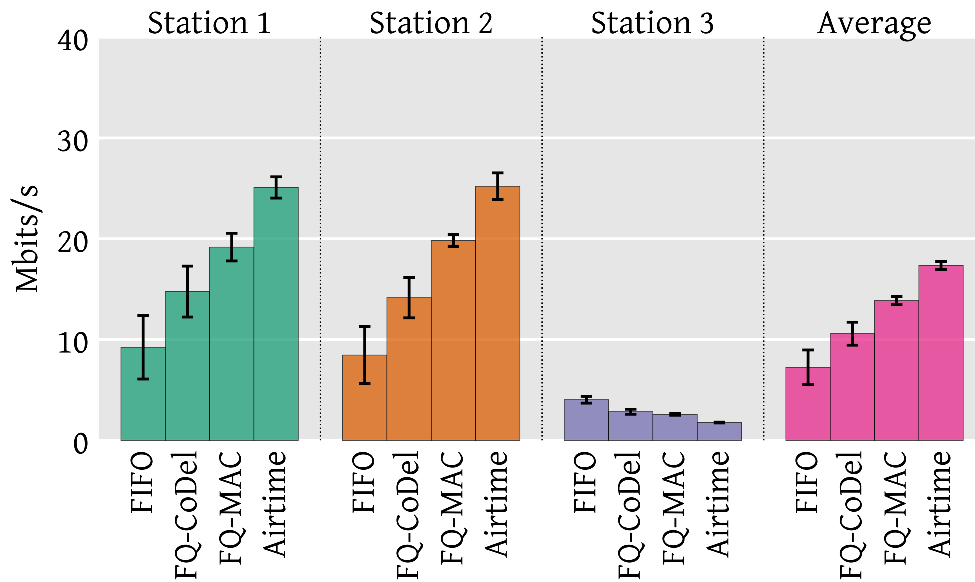


Figure 7: Throughput for TCP download traffic (to clients).

4.1.4 The sparse station optimisation

To evaluate the impact of the sparse station optimisation, we add a fourth station to our experiments which receives only a ping flow, but no other traffic, while the other stations receive bulk traffic as above. We measure the latency to this extra station both with and without the sparse station optimisation. The results of this are shown in Figure 8. For both UDP and TCP download traffic, the optimisation achieves a small, but consistent, improvement: The round-trip latency to the fourth station is reduced by 10 to 15% (in the median) when the optimisation is in place.

4.1.5 Scaling to more stations

While the evaluations presented in the previous sections have shown that our modifications work as planned, and that they provide a substantial benefit in a variety of scenarios, one question is left unanswered – does the solution scale to more stations? To answer this, we arranged for an independent third party to repeat a subset of our tests in their testbed, which features an access point and 30 clients. The nodes are all embedded wireless devices from a commercial vendor that bases its products on the OpenWrt/LEDE open-source router platform, running a LEDE firmware development snapshot from November 2016.

In this setup, one of the 30 clients is artificially limited to only transmit at the lowest possible rate (1 Mbps, i.e. disabling HT mode), while the others are configured to select their rate in the usual way, on a HT20 channel in the 2.4 Ghz band. One of the 29 "fast" clients only receives ping traffic, leaving 28 stations to contend with the slow 1 Mbps station for airtime and bandwidth.

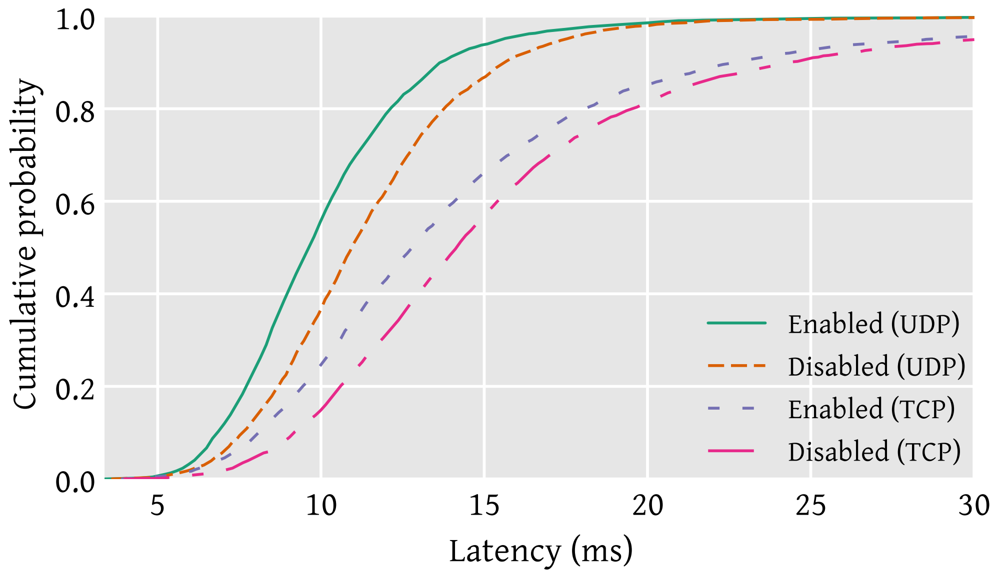
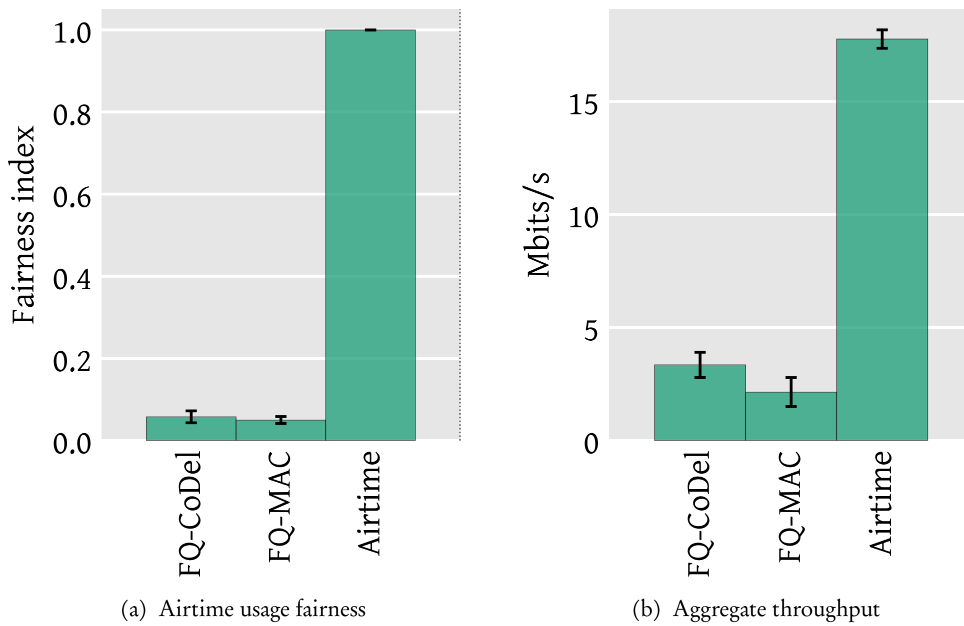


Figure 8: The effects of the sparse station optimisation.



(a) Airtime usage fairness (b) Aggregate throughput

Figure 9: Aggregate results for the 30 stations TCP test.

In this environment, our downstream TCP experiment presented above was repeated, with the difference that each test was run for five minutes, but with only five repetitions, and without the FIFO test case. A subset of these results are shown in figures 9 and 10. From this experiment, we make several observations:

1. When the slow station is at this very low rate, it manages to grab around two thirds of the available airtime, even with 28 other stations to compete with. However, our airtime fairness scheduler manages to achieve completely fair sharing of airtime between all 29 stations. This is reflected in the fairness index as seen in Figure 9a.
2. As seen in Figure 9b, total throughput goes from a mean of 3.3 Mbps for the FQ-CoDel case to 17.7 Mbps with the airtime scheduler. That is, the relative throughput gain with airtime fairness is 5.4x in this scenario.
3. As can be expected, with the airtime fairness scheduler, the latency to the fast stations is improved with the increased throughput (Figure 10, orange lines). However, the latency to the slow station increases by an order of magnitude in the median, as it is throttled to stay within its fair share of the airtime (Figure 10, dotted green line). Overall, the *average* latency to all stations is improved by a factor of two (not shown on the figure).
4. With 30 stations, we see the sparse station optimisation being even more effective; in this scenario it reduces latency to the sparse station by a factor of two (not shown in the figures; see the online appendix).

Finally, we verify the in-kernel airtime measurement against a tool developed by the same third party that measures airtime from captures taken with a monitor device. We find that the two types of measurements agree to within 1.5%, on average.

4.2 Effects on real-world application performance

In the previous section we evaluated our solution in a number of scenarios that verify its correct functioning and quantify its benefits. In this section we expand on that validation by examining how our modifications affect performance of two important real-world applications – VoIP and web browsing.

4.2.1 VoIP

VoIP is an important latency-sensitive application which it is desirable to have working well over WiFi, since that gives mobile handsets the flexibility of switching between WiFi and cellular data as signal conditions change. To evaluate our modifications in the context of VoIP traffic, we measure VoIP performance when mixed with bulk traffic. As in Section 4.1.4 we include a virtual station as another fast station, and so these scenarios have three fast

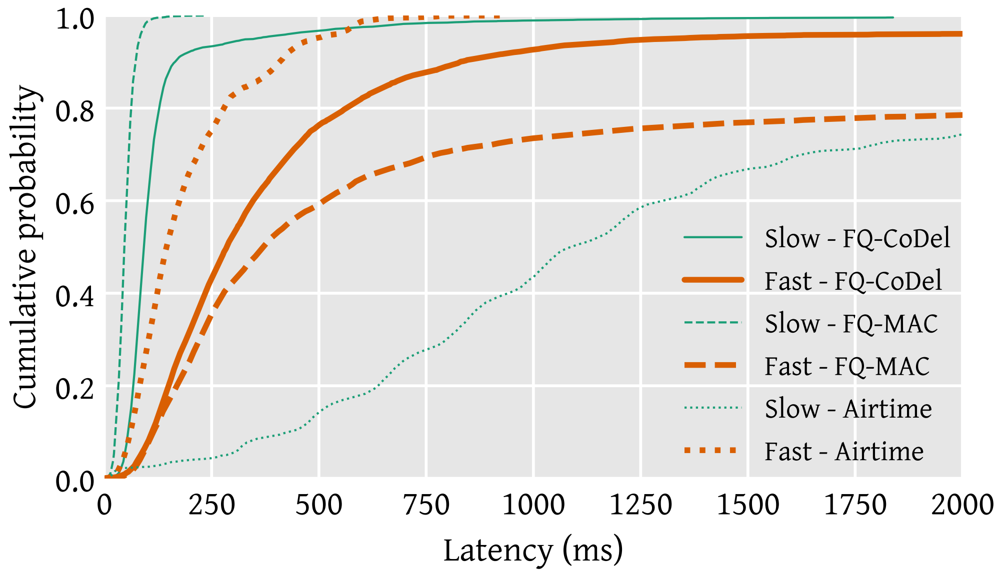


Figure 10: Latency for the 30 stations TCP test.

stations. Due to space constraints, we only include the case where the slow station receives both VoIP traffic *and* bulk traffic, while the fast stations receive bulk traffic. The other cases show similar relative performance between the different queue management schemes.

The QoS markings specified in the 802.11e standard can be used to improve the performance of VoIP traffic, and so we include this aspect in our evaluation. 802.11e specifies four different QoS levels, of which voice (VO) has the highest priority. Packets transmitted with this QoS marking gets both queueing priority and a shorter contention window, but cannot be aggregated. This difference can dramatically reduce the latency of the traffic, at a cost in throughput because of the lack of aggregation. We repeat the voice experiments in two variants – one where the VoIP packets are sent as best effort (BE) traffic, and one where they are put into the high-priority VO queue. We also repeat the tests with a baseline one-way delay of both 5 ms and 50 ms.

To serve as a metric of voice quality, we calculate an estimate of the Mean Opinion Score (MOS) of the VoIP flow according to the E-model specified in the ITU-T G.107 recommendation [17]. This model can predict the MOS from a range of parameters, including the network conditions. We fix all audio and codec related parameters to their default values and calculate the MOS estimate based on the measured delay, jitter and packet loss. The model gives MOS values in the range from 1 – 4.5.

Table 2 shows the results. This shows that throughput follows the trends shown in previous tests, as expected. Also as expected, the FIFO and FQ-CoDel cases have low MOS values when the voice traffic is marked as BE, and higher values when using the VO queue. However, both the FQ-MAC and airtime fairness schemes achieve *better* MOS values with best-effort traffic than the unmodified kernel does with VO-marked traffic. In the FQ-MAC and

Table 2: MOS values and total throughput when using different QoS markings for VoIP traffic. Data for 5 ms and 50 ms baseline one-way delay.

		5 ms		50 ms	
	QoS	MOS	Thrp	MOS	Thrp
FIFO	VO	4.17	27.5	4.13	21.6
	BE	1.00	28.3	1.00	22.0
FQ-CoDel	VO	4.17	25.5	4.08	15.2
	BE	1.24	23.6	1.21	15.1
FQ-MAC	VO	4.41	39.1	4.38	28.5
	BE	4.39	43.8	4.37	34.0
Airtime	VO	4.41	56.3	4.38	49.8
	BE	4.39	57.0	4.37	49.7

airtime cases, using the VO queue still gives a slightly better MOS score than using the BE queue does; but the difference is less than half a percent. This is an important improvement, because it means that with our modifications, applications can rely on excellent real-time performance even when unable to control DiffServ markings, as well as when the markings are removed in transit.

4.2.2 Web

Another important real-world application is web traffic. To investigate the impact of our modifications on this, we measure page load time (PLT) with emulated web traffic. Our test client mimics the common web browser behaviour of fetching multiple requests in parallel over four different TCP connections. We simply measure the total time to fetch a web site and all its attached resources (including the initial DNS lookup) for two different pages – a small page (56 KB data in three requests) and a large page (3 MB data in 110 requests). We run the experiments in two scenarios. One where a fast station fetches the web sites while the slow station runs a simultaneous bulk transfer, to emulate the impact of a slow station on the browsing performance of other users on the network. And another scenario where the slow station fetches the web sites while the fast stations run simultaneous bulk transfers, to emulate the browsing performance of a slow station on a busy network.

The results for the fast station are seen in Figure 11. Fetch times decrease from the FIFO case as the slowest to the airtime fair FQ case as the fastest. In particular, there is a an order-of-magnitude improvement when going from FIFO to FQ-CoDel, which we attribute to the corresponding significant reduction in latency seen earlier.

When the slow station is fetching the web page, adding airtime fairness increases page load time by 5 – 10%. This is as expected since in this case the

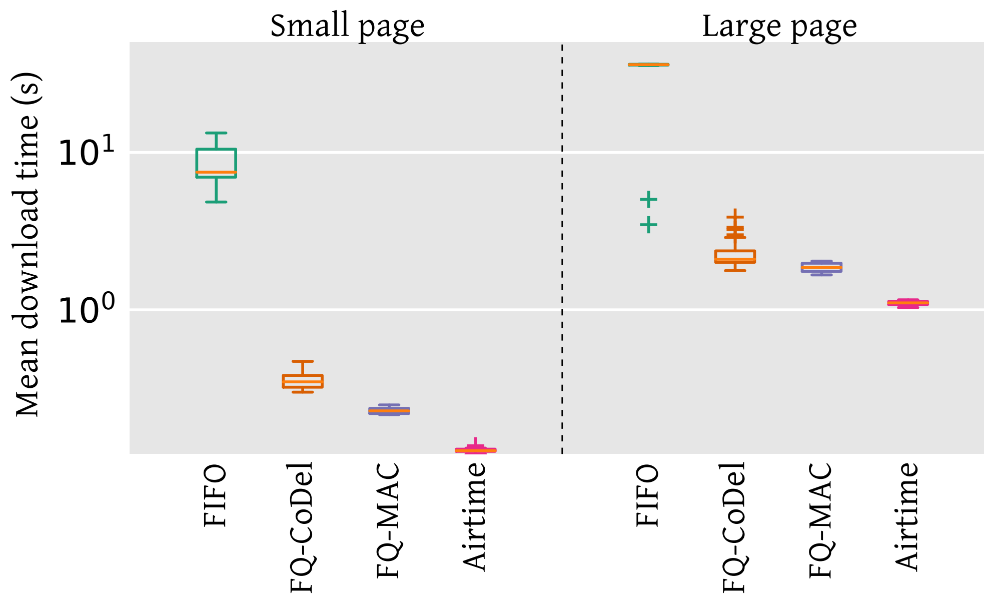


Figure 11: HTTP page fetch times for a fast station (while the slow station runs a bulk transfer). Note the log scale - the fetch time for the large page is 35 *seconds* for the FIFO case.

slow station is being throttled. The graph for this can be found in the online appendix.

4.3 Summary

Our evaluation shows that our modifications achieve their design goal. We eliminate bufferbloat and the 802.11 performance anomaly, and achieve close to perfect airtime fairness even when station rates vary; and our solution scales successfully as more clients are added. We improve total throughput by up to a factor of five and reduce latency under load by up to an order of magnitude. We also achieve close to perfect airtime fairness in a scenario where traffic is mixed between upstream and downstream flows from the different stations. Finally, the optimisation that prioritises sparse stations achieves a consistent improvement in latency to stations that only receive a small amount of traffic.

In addition, we show that our modifications give significant performance increases for two important real-world applications – VoIP and web traffic. In the case of VoIP, we manage to achieve better performance with best effort traffic than was achievable with traffic marked as Voice according to the 802.11e QoS standard. For web traffic we achieve significant reductions in page load times.

Finally, even though our evaluation scenario only features a limited number of stations, we have sought to represent a challenging scenario, by having high congestion rates and a large difference between the station rates. As such, we believe that it serves well as a validation of the effects. In addition, the feedback

we have received from users of the code indicates that our solution works well in a variety of deployments.

5 Related work

There have been several previous studies on bufferbloat and its mitigations (e.g. [5, 6]), but only a few that deal with the problem in a WiFi-specific context. [1] and [6] both feature a WiFi component in a larger evaluation of bufferbloat mitigation techniques and show that while these techniques can help on a WiFi link, the lower-level queueing in the WiFi stack prevents a full solution of the problem in this space. [7] draws similar conclusions while looking at buffer sizing (but only mentions AQM-based solutions briefly). Finally, [18] touches upon congestion at the WiFi hop and uses different queueing schemes to address it, but in the context of a centralised solution that also seek to control fairness in the whole network. None of these works actually provide a solution for bufferbloat at the WiFi link itself, as we present here.

Several different schemes to achieve fairness based on modifying the contention behaviour of nodes are presented in [4, 9, 19–22]. [9] and [19] both propose schemes that use either the 802.11e TXOP feature to allocate equal-length to all stations, or scaling of the contention window by the inverse of the transmission rate to achieve fairness. [4] develops an analytical model to predict the values to use for a similar scaling behaviour, which is also verified in simulation. [20] presents a modified contention behaviour that can lower the number of collisions experienced, but they do not verify the effect of their schemes on airtime fairness. [21] proposes a modification to the DCF based on sensing the idle time of the channel scaling CW_{min} with the rate to achieve fairness. Finally, [22] proposes a scheme for airtime fairness that runs several virtual DCF instances per node, scaling the number of instances with the rate and channel properties.

Achieving fairness by varying the transmission size is addressed in [10, 23, 24]. The former two predate the aggregation features of 802.11n and so [23] proposes to scale the packet size downwards by varying the MTU with the transmission rate. [24] goes the other way and proposes a scheme where a station will burst packets to match the total transmission length of the previous station that was heard on the network. Finally, [10] uses the two-level aggregation feature of 802.11n and proposes a scheme to dynamically select the optimal aggregation size so all transmissions take up the same amount of time.

Turning to schedulers, [25] and [16] both propose schedulers which work at the access point to achieve airtime fairness, the former estimating the packet transmission time from channel characteristics, and the latter measuring it after transmission has occurred. [26] proposes a solution for wireless mesh networks, which leverages routing metrics to schedule links in a way that ensures fairness. Finally, [27] proposes a scheme to scale the queue space at the access point based on the BDP of the flows going through the access point. Our solution is closest to [16], but we improve upon it by increasing accuracy

and reducing implementation difficulty, while adding an important latency-reducing optimisation for sparse stations, as was described in Section 3.2.

A few proposals fall outside the categories above. [28] proposes a TCP congestion control algorithm that uses information about the wireless conditions to cap the TCP window size of clients to achieve fairness. Finally, there are schemes that sidestep the fairness problems of the 802.11 MAC and instead replace it entirely with TDMA scheduling. [29] proposes a scheme for TDMA scheduling in a mesh network that ensures fair bandwidth allocation to all connecting clients, and [30] implements a TDMA transmission scheme for infrastructure WiFi networks.

6 Conclusion

We have developed a novel two-part solution to two large performance problems affecting WiFi – bufferbloat and the 802.11 performance anomaly. The solution consists of a new integrated queueing scheme tailored specifically to eliminate bufferbloat in WiFi, which reduces latency under load by an order of magnitude. Leveraging the queueing structure, we have developed a deficit-based airtime fairness scheduler that works at the access point with no client modifications, and achieves close to perfect fairness in all the evaluated scenarios, increasing total throughput by up to a factor of 5.

Our solution reduces implementation complexity and increases accuracy compared to previous work, and has been accepted into the mainline Linux kernel, making it deployable on billions of Linux-based devices.

7 Acknowledgements

We would like to thank Sven Eckelmann and Simon Wunderlich for their work on independently verifying our implementation. Their work was funded by Open Mesh Inc, who also supplied their test hardware. We would also like to thank Felix Fietkau, Tim Shepard, Eric Dumazet, Johannes Berg, and the numerous other contributors to the Make-Wifi-Fast and LEDE projects for their insights, review and contributions to many different iterations of the implementation.

Portions of this work were funded by Google Fiber and by the Comcast Innovation Fund, and parts of the infrastructure was sponsored by Lupin Lodge.

References

[1] T. Høiland-Jørgensen, P. Hurtig, and A. Brunstrom, "The Good, the Bad and the WiFi: Modern AQMs in a residential setting," *Computer Networks*, vol. 89, pp. 90–106, Oct. 2015.

[2] M. Heusse *et al.*, "Performance anomaly of 802.11 b," in *INFOCOM 2003. Twenty-Second Annual Joint Conference of the IEEE Computer and Communications. IEEE Societies*, vol. 2. IEEE, 2003, pp. 836–843.

[3] G. Tan and J. V. Guttag, "Time-based fairness improves performance in multi-rate WLANs." in *USENIX Annual Technical Conference, General Track*, 2004, pp. 269–282.

[4] T. Joshi *et al.*, "Airtime fairness for IEEE 802.11 multirate networks," *IEEE Transactions on Mobile Computing*, vol. 7, no. 4, pp. 513–527, Apr 2008.

[5] G. White, "Active queue management algorithms for DOCSIS 3.0: A simulation study of CoDel, SFQ-CoDel and PIE in DOCSIS 3.0 networks," Cable Television Laboratories, Inc., Tech. Rep., 2013.

[6] N. Khademi, D. Ros, and M. Welzl, "The new AQM kids on the block: Much ado about nothing?" Oslo University, Tech. Rep. 434, 2013.

[7] A. Showail, K. Jamshaid, and B. Shihada, "Buffer sizing in wireless networks: challenges, solutions, and opportunities," *IEEE Communications Magazine*, vol. 54, no. 4, pp. 130–137, Apr 2016.

[8] M. Laddomada *et al.*, "On the throughput performance of multirate IEEE 802.11 networks with variable-loaded stations: analysis, modeling, and a novel proportional fairness criterion," *IEEE Transactions on Wireless Communications*, vol. 9, no. 5, pp. 1594–1607, May 2010.

[9] L. B. Jiang and S. C. Liew, "Proportional fairness in wireless LANs and ad hoc networks," in *Wireless Communications and Networking Conference, 2005 IEEE*, vol. 3. IEEE, 2005, pp. 1551–1556.

[10] M. Kim, E.-C. Park, and C.-H. Choi, "Adaptive two-level frame aggregation for fairness and efficiency in IEEE 802.11n wireless LANs," *Mobile Information Systems*, 2015.

[11] T. Y. Arif and R. F. Sari, "Throughput Estimates for A-MPDU and Block ACK Schemes Using HT-PHY Layer," *Journal of Computers*, vol. 9, no. 3, Mar. 2014.

[12] Adam Belay *et al.*, "IX: A protected dataplane operating system for high throughput and low latency," in *Proceedings of the 11th USENIX Symposium on Operating System Design and Implementation (OSDI '14)*, 10 2014, pp. 49–65.

[13] T. Szigeti, J. Henry, and F. Baker, "Mapping Diffserv to IEEE 802.11," RFC 8325 (Proposed Standard), RFC Editor, Feb. 2018.

[14] T. Høiland-Jørgensen *et al.*, "The Flow Queue CoDel Packet Scheduler and Active Queue Management Algorithm," RFC 8290 (Experimental), RFC Editor, Jan. 2018.

[15] M. Shreedhar and G. Varghese, "Efficient fair queuing using deficit round-robin," *IEEE/ACM Transactions on Networking*, vol. 4, no. 3, pp. 375–385, Jun. 1996.

[16] R. G. Garroppo *et al.*, "Providing air-time usage fairness in IEEE 802.11 networks with the deficit transmission time (DTT) scheduler," *Wireless Networks*, vol. 13, no. 4, pp. 481–495, Aug 2007.

[17] "The E-model: a computational model for use in transmission planning," ITU-T, Tech. Rep. G.107, Jun. 2015.

[18] K. Cai *et al.*, "Wireless Unfairness: Alleviate MAC Congestion First!" in *Proceedings of the Second ACM International Workshop on Wireless Network Testbeds, Experimental Evaluation and Characterization*, ser. WinTECH '07. ACM, 2007, pp. 43–50.

[19] P. Lin, W.-I. Chou, and T. Lin, "Achieving airtime fairness of delay-sensitive applications in multirate IEEE 802.11 wireless LANs," *IEEE Communications Magazine*, vol. 49, no. 9, pp. 169–175, 2011.

[20] L. Sanabria-Russo *et al.*, "Future evolution of CSMA protocols for the IEEE 802.11 standard," in *2013 IEEE International Conference on Communications Workshops (ICC)*. IEEE, 2013, pp. 1274–1279.

[21] M. Heusse *et al.*, "Idle Sense: An Optimal Access Method for High Throughput and Fairness in Rate Diverse Wireless LANs," in *Proceedings of the 2005 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. ACM, 2005, pp. 121–132.

[22] M. A. Yazici and N. Akar, "Running Multiple Instances of the Distributed Coordination Function for Air-Time Fairness in Multi-Rate WLANs," *IEEE Transactions on Communications*, vol. 61, no. 12, pp. 5067–5076, Dec. 2013.

[23] J. Dunn *et al.*, "A practical cross-layer mechanism for fairness in 802.11 networks," in *First International Conference on Broadband Networks (BroadNets 2004)*. IEEE, 2004, pp. 355–364.

[24] T. Razafindralambo *et al.*, "Dynamic packet aggregation to solve performance anomaly in 802.11 wireless networks," in *Proceedings of the 9th ACM international symposium on Modeling analysis and simulation of wireless and mobile systems*. ACM, 2006, pp. 247–254.

[25] K. Gomez *et al.*, "On efficient airtime-based fair link scheduling in IEEE 802.11-based wireless networks," in *2011 IEEE 22nd International Symposium on Personal, Indoor and Mobile Radio Communications*, Sep. 2011, pp. 930–934.

[26] R. Riggio, D. Miorandi, and I. Chlamtac, "Airtime Deficit Round Robin (ADRR) packet scheduling algorithm," in *2008 5th IEEE International Conference on Mobile Ad Hoc and Sensor Systems*, Sep. 2008, pp. 647–652.

[27] D. Kliazovich *et al.*, "Queue Management Mechanism for 802.11 Base Stations," *IEEE Communications Letters*, vol. 15, no. 7, pp. 728–730, Jul. 2011.

[28] K. Kashibuchi, A. Jamalipour, and N. Kato, "Channel Occupancy Time Based TCP Rate Control for Improving Fairness in IEEE 802.11 DCF," *IEEE Transactions on Vehicular Technology*, vol. 59, no. 6, pp. 2974–2985, Jul. 2010.

[29] N. Ben Salem and J.-P. Hubaux, "A fair scheduling for wireless mesh networks," in *First IEEE Workshop on Wireless Mesh Networks (WiMesh)*, 2005.

[30] W. Torfs and C. Blondia, "TDMA on commercial of-the-shelf hardware: Fact and fiction revealed," *AEU-International Journal of Electronics and Communications*, vol. 69, no. 5, pp. 800–813, 2015.

Paper VI

PoliFi
Airtime Policy Enforcement for WiFi

Under Submission

"Honesty is the best policy."
Benjamin Franklin

PoliFi

Airtime Policy Enforcement for WiFi

Toke Høiland-Jørgensen, Per Hurtig and Anna Brunstrom
{toke.hoiland-jorgensen, per.hurtig, anna.brunstrom}@kau.se

Abstract

As WiFi grows ever more popular, airtime contention becomes an increasing problem. One way to alleviate this is through network policy enforcement. Unfortunately, WiFi lacks protocol support for configuring policies for its usage, and since network-wide coordination cannot generally be ensured, enforcing policy is challenging.

However, as we have shown in previous work, an access point can influence the behaviour of connected devices by changing its scheduling of transmission opportunities, which can be used to achieve airtime fairness. In this work, we show that this mechanism can be extended to successfully enforce airtime usage policies in WiFi networks. We implement this as an extension our previous airtime fairness work, and present PoliFi, the resulting policy enforcement system.

Our evaluation shows that PoliFi makes it possible to express a range of useful policies. These include prioritisation of specific devices; balancing groups of devices for sharing between different logical networks or network slices; and limiting groups of devices to implement guest networks or other low-priority services. We also show how these can be used to improve the performance of a real-world DASH video streaming application.

1 Introduction

WiFi is increasingly becoming the ubiquitous connectivity technology in homes as well as in enterprises. The ability for anyone to set up an access point and connect any device to it is one of the driving factors behind this increase of popularity. However, increased popularity also means increased contention for resources as more devices are deployed.

Since no two devices can transmit at the same time on a given frequency, the sparse resource that determines performance in WiFi networks is the time

spent transmitting, also known as airtime usage. The 802.11 Media Access Control (MAC) protocol used in WiFi networks does not, in itself, guarantee a fair usage of this sparse resource. In fact it is well known that devices transmitting at lower rates can use more than their fair share of the airtime [1].

One way to improve performance of a network under contention is to apply different policies to different devices on the network, which works best if applied directly to the sparse resource instead of a proxy such as byte-level throughput. However, WiFi is decentralised at the protocol level, and thus lacks protocol support for enforcing policies on airtime usage. Fortunately, it turns out that in the common infrastructure deployment scenario, the access point can exert quite a bit of influence on the transmission behaviour of clients, or *stations*, as they are commonly called. In previous work, we have shown that this makes it possible to achieve airtime fairness between stations in a WiFi network by making appropriate scheduling decisions at the AP [2]. Given such a mechanism to enforce fairness, a natural question is whether it can be extended to express different capacity sharing policies. In this work we answer this question in the affirmative, in the form of a workable solution to airtime policy enforcement in WiFi, which we have named *PoliFi*.

The number of possible policies one might want to express is all but infinite. Therefore, to focus our discussion, we define the following three representative policy use cases:

1. Prioritising devices. It should be possible to configure one or more devices to receive a higher share of network resources than other devices on the network.

2. Balancing device groups. In this use case, the network should be configured to share the available resources between groups of devices in a given way. For instance, this could be used to implement the "network slicing" concept often seen in 5G architectures [3].

3. Limiting groups of devices, as a way to implement a lower-priority service, such as a guest network. Here, a logical group of devices should be limited to a *maximum* capacity share.

PoliFi makes it possible for the user to express all of these policies, and we also solve a number of practical implementation issues to achieve its realisation. Our design is a two-part solution including (a) a userspace daemon that is configurable and implements the higher-level policy decisions, and (b) a scheduler integrated with the (Linux) operating system WiFi stack.

The rest of this paper presents PoliFi in detail, and is structured as follows: Section 2 summarises related work. Section 3 describes our design, with a performance analysis presented in Section 4. Finally, Section 5 concludes.

2 Related work

Network policies are, in general, nothing new. For instance, standardisation of different traffic classes has occurred in the form of the DiffServ framework [4].

In the WiFi world, the 802.11e standard defines different priority levels, which can be mapped to DiffServ code points [5]. However, this is all related to applying policies to different types of traffic, whereas PoliFi deals with realising different capacity sharing policies between devices on the same network at the airtime usage level. As such, PoliFi is orthogonal so DiffServ, 802.11e and other traffic class policy mechanisms.

As mentioned above, PoliFi is an extension of our previous work implementing an airtime fairness enforcement mechanism in Linux [2]. Compared to this previous work, PoliFi adds the policy enforcement component, and also generalises the mechanism by moving it out of the device drivers and into the common WiFi subsystem in Linux, thus making it applicable to more device drivers.

The concept of airtime policy enforcement appears in the concept of *network slicing*, which is an important part of the upcoming 5G mobile network architecture [3]. Network slicing involves splitting up a network into several virtual parts that are conceptually isolated from one another, which is a form of policy enforcement. A description of how to achieve network slicing in WiFi networks is given in [6], which corresponds roughly to our second use-case. The authors implement a prototype in simulation. Our mechanism builds on the same basic concept of computing per-device weights from group weights, but we solve a number of issues that prevent it from being implemented on real hardware. In addition, [6] only covers the second of our three policy use-cases.

Another approach to splitting a wireless network into multiple parts is presented in [7], which describes a scheme where a separate software router is installed in the access point. This software router queues packets and enforces capacity sharing. However, the capacity sharing is implemented at the bandwidth level which, as mentioned above, is not the sparse resource in a WiFi network.

A description of a scheme for network slicing in a home network is described in [8]. The authors describe a design that uses Software Defined Networking (SDN) to split a home network into different parts, but do not discuss any mechanism for how the sharing is achieved.

Finally, some enterprise APs offer features related to airtime fairness and policy configuration, e.g., [9]. Unfortunately, no technical description of how these policies are enforced is generally available, which prevents us from comparing them to our solution.

3 The PoliFi Design

We have designed PoliFi as a two-part solution, where a user-space daemon is configured by the user, and in turn configures a scheduling mechanism in the kernel. In this section, we describe our design in detail. A diagram of the design is shown in Figure 1. We begin by describing the user space daemon that configures the policies. Following this, we describe how the weighted Deficit Round-Robin (DRR) scheduling mechanism is used to achieve the

desired policies, and finally we describe how the mechanism is integrated into the Linux kernel WiFi stack.

3.1 Userspace Policy Daemon

We implement the userspace policy daemon as part of the *hostapd* access point management daemon. This is the daemon responsible for configuring wireless devices in access point mode in Linux. This means it already implements policies for other aspects of client behaviour (such as authentication), which makes it a natural place to implement airtime policy as well.

The module we have added to hostapd can be configured in three modes, corresponding to the three use cases described in the introduction: static mode, dynamic mode and limit mode. The user can configure each of these modes per physical WiFi domain, and assign parameters for individual stations (based on their MAC addresses), or for entire Basic Service Sets (BSSes). The latter is a natural grouping mechanism, since this corresponds to logical networks configured on the same device (e.g., a primary and a guest network). However, extending the design to any other logical grouping mechanism is straight forward.

In static mode, the daemon will simply assign static weights to stations when they associate to the access point. Weights can be configured for individual stations, while a default weight can be set for each BSS, which will be applied to all stations that do not have an explicit value configured. This implements the basic use case of assigning higher priorities to specific devices, but does not guarantee any specific total share.

The dynamic and limit modes work by assigning weights to each BSS, which are interpreted as their relative shares of the total airtime, regardless of how many stations they each have associated. Additionally, in limit mode, one or more BSSes can be marked as *limited*. BBSes that are marked as limited are not allowed to exceed their configured share, whereas no limitations are imposed on unmarked BSSes. Thus, dynamic mode implements the second use case, while limit mode implements the third.

For both dynamic mode and limit mode, the daemon periodically polls the kernel to discover which stations are currently active, using the queue backlog as a measure of activity, as discussed below. After each polling event, per-station weights are computed based on the number of active stations in each BSS, and these weights are configured in the kernel. The details of the weight computation, and how this is used to achieve the desired policy, is discussed in the next section. Selecting the polling frequency is a tradeoff between reaction time and system load overhead. The polling interval defaults to 100 ms, which we have found to offer good reaction times (see Section 4.2), while having a negligible overhead on our test system.

While our implementation is focused on the single access point case, where the access point enforces a single configured policy, the userspace daemon could just as well pull its policy configuration from a central cloud-based management service, while retaining the same policy enforcement mechanism.

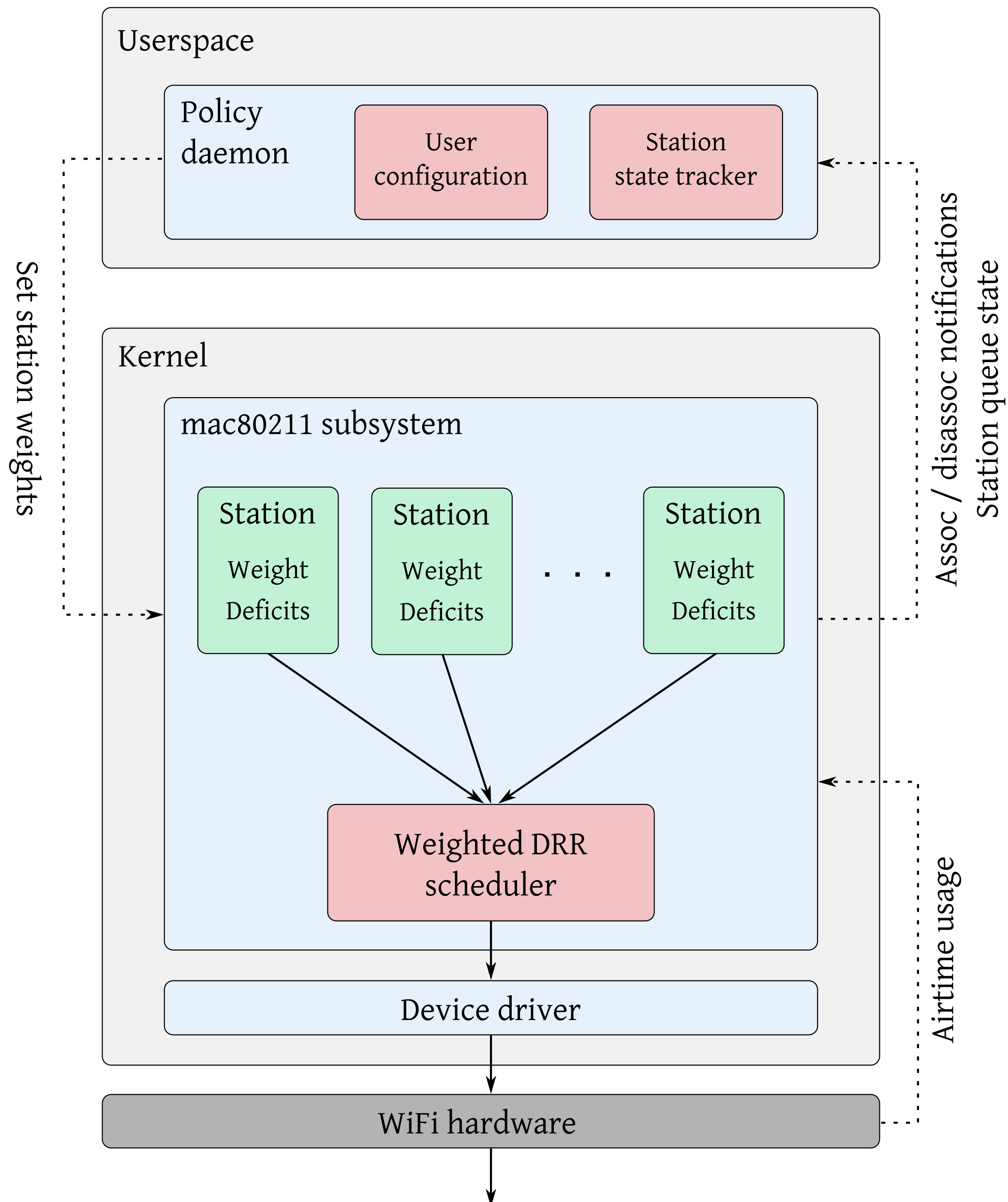


Figure 1: The high-level design of PoliFi. The kernel maintains data structures for every station, containing its current airtime deficits and configured weight. The scheduler uses this to decide which station to transmit to next. The hardware reports airtime usage on TX completion. The userspace daemon tracks the associated stations and their queue state, and updates the weights in the kernel based on user policy preferences.

3.2 Weighted Airtime DRR

The fairness mechanism that we are starting from (described in detail in [2]) is a Deficit Round-Robin scheduler, which operates by accounting airtime usage as reported by the WiFi hardware after a transmission has completed, and scheduling transmissions to ensure that the aggregate usage over time is the same for all active stations. Using the airtime information provided after transmission completes means that retransmissions can be accounted for, which improves accuracy especially for stations with low signal quality. Furthermore, as we have shown in our previous work, for TCP traffic we can provide fairness even for transmissions transfers coming *from* each station. This is achieved by accounting the airtime of received packets, which causes the scheduler to throttle the rate of TCP ACKs going back to the station.

3.2.1 Adding Weighted Scheduling

Given this effective airtime fairness scheduler, we can realise arbitrary division of the available capacity between different stations, by simply assigning them different scheduling weights. For the DRR scheduling algorithm employed by our scheduler, this is achieved by using different quantums per station. Thus, to apply this to airtime policy enforcement, we need to express the desired policy as a number of different service weights for each of the active devices.

The first use case is trivially expressed in terms of weights: simply assign the prioritised device a higher weight; for instance, to double its priority, assign it a weight of 2. The second use case, where capacity should be split between groups of devices has been covered in the network slicing use case described in [6]: each group is assigned a weight signifying its share relative to the other groups; from these group weights, each device in that group is assigned a weight computed by dividing the group weight with the number of active devices in that group.

The final use case requires limiting one or more groups of stations to a fixed share of the available capacity. This can be illustrated with the guest network use case, where an example policy could be that a guest network is not allowed to exceed 50% of the available capacity. If this policy is implemented as a fixed share between groups, however, a single station on the guest network would be able to get the same capacity as, say, five users of the regular network, which is not what we want. Thus, a different policy is needed: a group can be limited, and should have its weight adjusted only if it would get *more* than the configured share, not if it gets less. Thus, this becomes a two-step procedure that first assigns unit weights to all devices (which is the default when no policy is applied), and calculates whether or not this results in the limited group using more than its configured share of the airtime. If it does, a policy is computed in the same way as for the dynamic use case, which results in the limited group being assigned exactly its configured airtime share.

3.2.2 Computing the Weights

Having established that our desired policies can be expressed in terms of weights, we turn to the practical difficulties of applying this to a real WiFi system.

First, the approach outlined above assumes that we have knowledge of which stations are active at any given time. This might look trivial at first glance, since an access point needs to maintain some amount of state for all currently associated clients in any case. However, clients can be associated to an access point without sending or receiving any data, and thus without consuming any airtime. This means that association state in itself is not sufficient to ascertain the set of currently active clients. Fortunately, we have another piece of data: The queue backlog for each device. Monitoring the backlog gives us a straight-forward indicator for activity without having to monitor actual packet flows; we can simply consider any device that has had a non-zero queue backlog within a suitably short time span as active, and use that number in our calculations.

The second difficulty lies in the fact that we need to transform the total weights between groups of stations into weights for each individual station. As shown in [6], this is conceptually just a simple division. However, when implementing this in an operating system kernel, we are limited to integer arithmetic, which means that to get accurate weights, we need to ensure that the division works when confined to the integers. To achieve this, we first limit our configuration language to be expressed as integer weights between groups. Then, to ensure that we can divide these weights with the number of active stations, we multiply them by a suitable constant, chosen as follows:

We are given the set of groups I, where each group i has a configured group weight W_i and N_i active stations. We then define the multiplication constant $C = \prod_{i \in I} N_i$. Multiplying all group weights by this same constant maintains their relative ratio, and the choice of constant ensures that each group's weight can be divided by the number of active stations in that group. This gives us the following expression for the per-station weight for group i:

$$W_i^s = \frac{W_i C}{N_i} \tag{1}$$

The third issue we need to deal with is converting the weights to the per-station time quantums that are used in the scheduler, and which are expressed in microseconds. These should be kept at a reasonable absolute size, because larger weights result in coarser time granularity of the scheduler, making each scheduler round take longer and impacting latency of all devices in the network. We convert the calculated weights into final quantums by normalising them so they fall within a range of $100 - 1000\,\mu s$, but preferring smaller values if the ratio between the smallest and largest weight is more than $10\times$.

3.3 Kernel Airtime Scheduler

We implement the kernel part of PoliFi in the WiFi protocol subsystem of the Linux kernel (called *mac80211*). Our implementation builds on our previous airtime fairness scheduler, described in [2], which implemented a queueing system in this layer. In this queueing system, packets are assigned a Traffic ID (TID) before enqueue, and a separate queueing structure is created for each TID, of which there are 16 per station. These per-TID queues then form the basis of the scheduling of different stations. The queueing structure itself is based on the FQ-CoDel queue management scheme [10] and ensures flow isolation and low queueing latency.

While our previous implementation implemented queueing in the general WiFi layer, scheduling and tracking of each active station's airtime usage was still the responsibility of the driver, and so fairness and policy scheduling needed to be re-implemented by each driver. In PoliFi, we generalise the scheduler by moving parts of it into mac80211, where it can be leveraged by all device drivers, and specify a new API for device drivers to use. We then modify the *ath9k* device driver for Qualcomm Atheros 802.11n devices to use this API. The API moves the responsibility for deciding which TID to service next into mac80211, where before this decision was the responsibility of the device driver. The functions implementing and using the API are shown in Algorithm 5. The device driver runs the `schedule()` function, and asks mac80211 for the next TID queue to schedule, using the `get_next_tid()` API function. The driver then services this queue until no more packets can be scheduled (typically because the hardware queue is full, or the TID queue runs empty). After this, the driver uses the `return_tid()` API function to return the TID queue to the scheduler. A third API function, `account_airtime()`, allows the driver to register airtime usage for each station, which is typically done asynchronously as packets are completed or received.

Using this API, mac80211 has enough information to implement airtime policy enforcement using the weighted deficit scheduler approach described above. As for the previous airtime fairness scheduler in the driver, airtime deficits are kept separately for each of the four hardware QoS levels, to match the split of the hardware transmission queue scheduling. The algorithm is implemented as part of the `get_next_tid()` function as shown in Algorithm 6. To allow userspace to adjust the weights of each station, we also add new functions to the existing userspace API of the Linux WiFi stack (not shown here due to space constraints).

4 Evaluation

In this section we evaluate how effectively PoliFi is able to implement the desired policies. We examine steady state behaviour as well as the reaction time of the dynamic and limit modes with a changing number of active stations. To show how airtime policies can provide benefits for specific applications, we also include an DASH video streaming use case in our evaluations. We

Algorithm 6 Airtime fairness scheduler. The schedule function is part of the device driver and is called on packet arrival and on transmission completion. The account_airtime function is called by the driver when it receives airtime usage information on TX completion or packet reception.

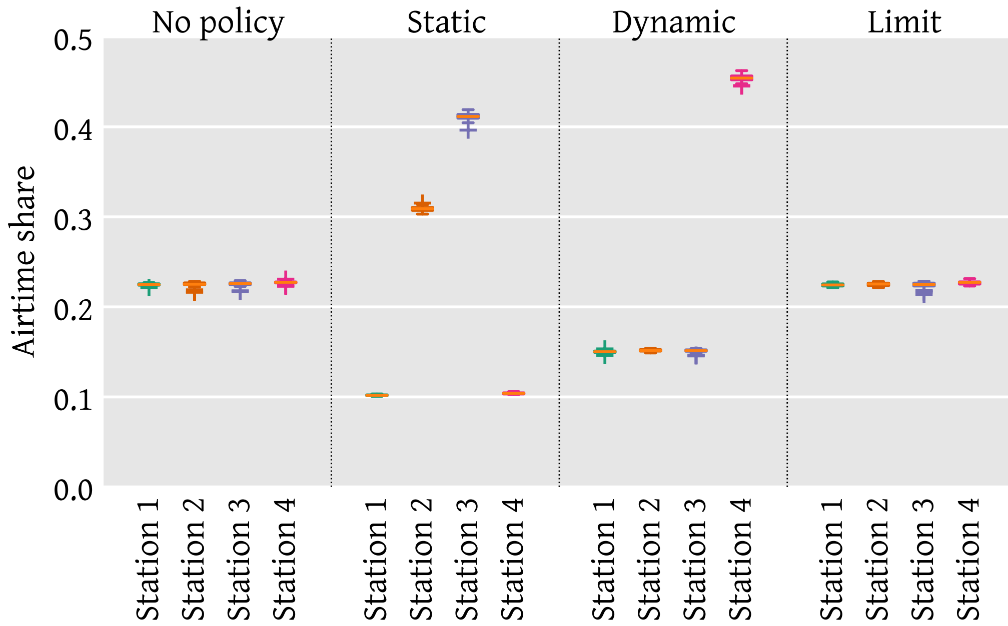
```
 1: function SCHEDULE(qoslvl)
 2:     tid ← GET_NEXT_TID(qoslvl)
 3:     BUILD_AGGREGATE(tid)
 4:     RETURN_TID(tid)
 5: function GET_NEXT_TID(qoslvl)
 6:     tid ← FIND_FIRST(active_tids, qoslvl)
 7:     stn ← tid.station
 8:     deficit ← stn.deficit[qoslvl]
 9:     if deficit ≤ 0 then
10:         stn.deficit[qoslvl] ← deficit + stn.weight
11:         LIST_MOVE_TAIL(tid, active_tids)
12:         restart
13:     LIST_REMOVE(tid, active_tids)
14:     return tid
15: function RETURN_TID(tid)
16:     if tid.queue is not empty then
17:         LIST_ADD_HEAD(tid, active_tids)
18:
19: function ACCOUNT_AIRTIME(tid, airtime)
20:     stn ← tid.station
21:     qoslvl ← tid.qoslvl
22:     stn.deficit[qoslvl] ← stn.deficit[qoslvl] − airtime
```

perform the experiments on our testbed with four WiFi devices. The details of our setup are omitted here due to space constraints, but are available in an online appendix.[22]

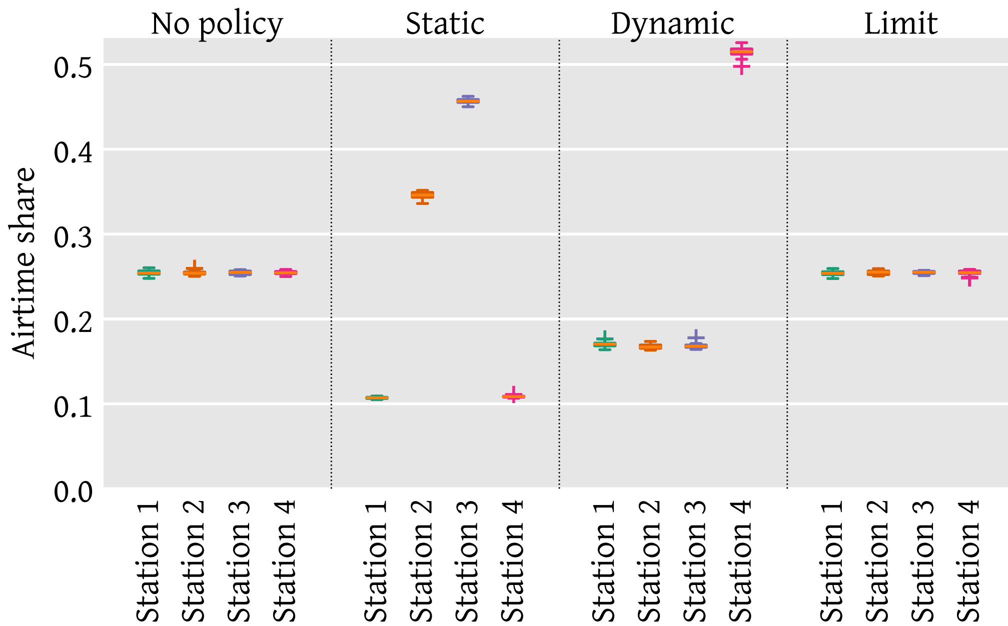
4.1 Steady state measurements

The steady state tests consist of running a bulk flow (either UDP or TCP) to each of four stations associated to the access point running PoliFi. Three of the stations are associated to one BSS on the access point, while the fourth is on a separate BSS. These two BSSes are the groups the algorithm balances in dynamic and limit mode. Both groups are given equal weights, meaning that they should receive the same total airtime share. When testing the limit mode use case, the BSS with only a single station in it is set to *limited*, which in this case means that its natural airtime share is less than the configured share, and thus that no limiting is necessary to enforce the configured policy. We test this to ensure that the algorithm correctly allows the group that is not marked as limited to exceed its configured airtime share.

[22] See `https://www.cs.kau.se/tohojo/polifi`.

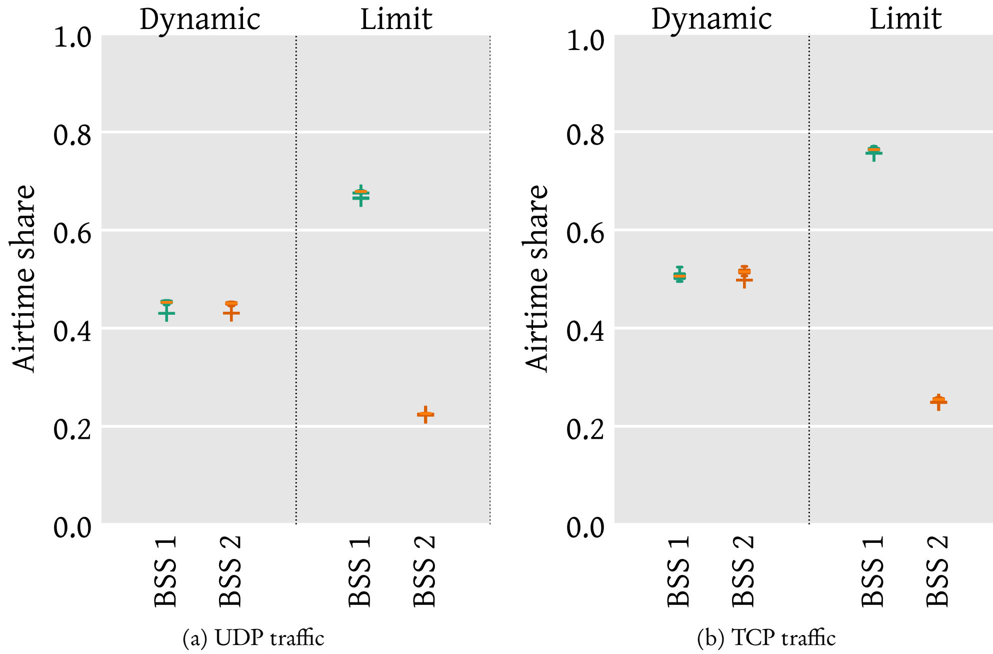


(a) UDP traffic



(b) TCP traffic

Figure 2: Aggregate airtime usage share of four stations, over a 30-second bulk transfer. Graph columns correspond to the different policy modes. In static mode stations 2 and 3 are assigned weights of 3 and 4, respectively. In dynamic and limit mode, stations 1-3 are on one BSS while station 4 is on another; both BSSes have the same weight, and the second BSS is set to limited. The plots are box plots of 30 test runs, but look like lines due to the low variation between runs.



(a) UDP traffic

(b) TCP traffic

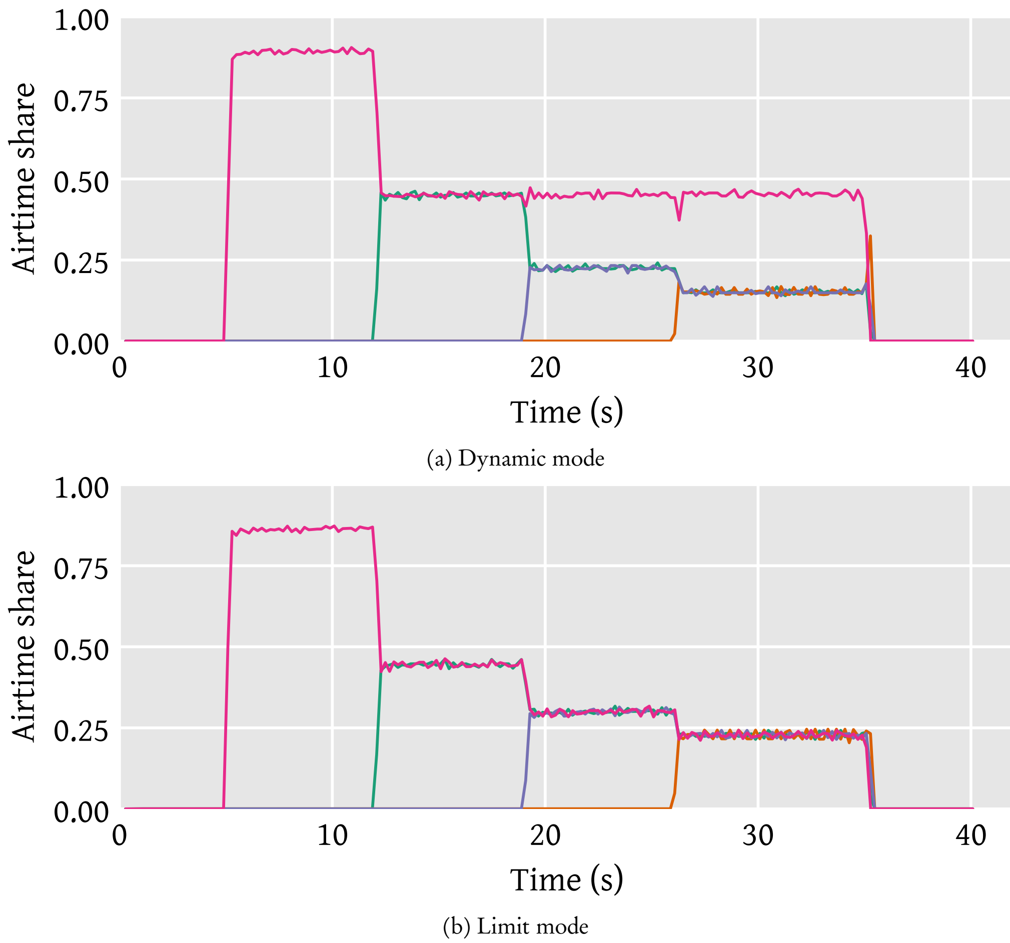
Figure 3: Aggregate airtime usage of the two BSSes, for the same test as that shown in Figure 2.

The aggregate airtime usage of the stations and BSSes is seen in figures 2 and 3, respectively. With no policy configured, the scheduler simply enforces fairness between the active stations. In the static policy mode, relative weights of 3 and 4 are assigned to stations 2 and 3, respectively. These weights are clearly reflected in the airtime shares achieved by each station in the second column of the graphs in Figure 2, showing that static policy assignment works as designed.

Turning to the group modes, Figure 3 shows the aggregate airtime for each of the two configured BSSes. In dynamic mode, the scheduler enforces equal sharing between the two BSSes, which translates to the single station in BSS 2 getting three times as much airtime as the other three, as is seen in the third column of Figure 2. In limit mode, BSS 2 is limited to at most half of the airtime, but because there is only one station connected to it, its fair share is already less than the limit, and so this corresponds to the case where no policy is enforced. Thus, the tests show that the scheduler successfully enforces the configured policies for all three use cases.

4.2 Dynamic measurements

To evaluate the reaction time of the scheduler as station activity varies, we perform another set of UDP tests where we start the flows to each of the stations five seconds apart. We perform this test for the dynamic and limit modes, as these are the cases where the scheduler needs to react to changes in station activity.



(a) Dynamic mode

(b) Limit mode

Figure 4: Airtime usage over time with changing number of active stations, in dynamic and limit mode. UDP flows to each station start 5 seconds apart. The purple station (starting first) is on one BSS, while the remaining three stations are on the other BSS.

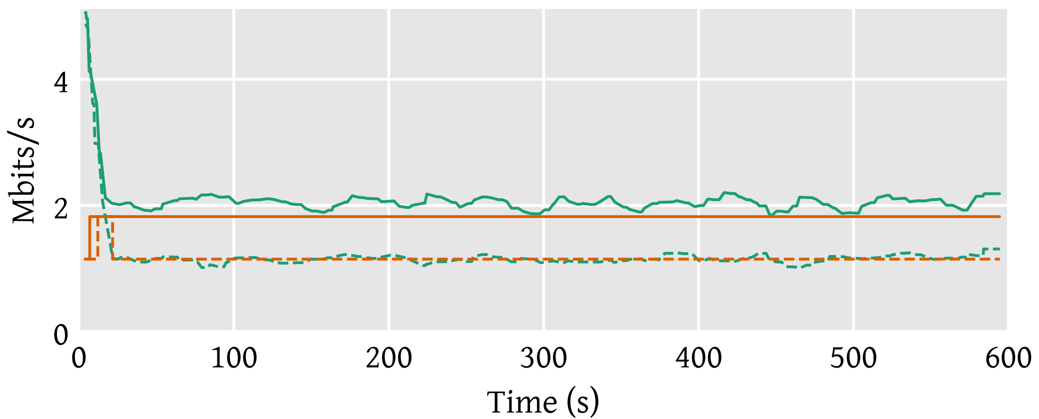


Figure 5: DASH video throughput with prioritisation (solid lines) and without (dashed lines). The straight lines (orange) show the video bitrate picked by the player, while the others show the actual data stream goodput.

The results of this dynamic test is shown in Figure 4 as time series graphs of airtime share in each 200 ms measurement interval. The station that starts first is Station 4 from the previous graphs, i.e., the station that is on BSS 2. In dynamic mode, as seen in Figure 4a, the first station is limited to half the available airtime as soon as the second station starts transmitting. And because the two groups are set to share the airtime evenly, as more stations are added, the first station keeps using half the available airtime, while the others share the remaining half.

In limit mode, as we saw before, the airtime shares of each of the four stations correspond to their fair share. This is also seen in Figure 4b, where all stations share the airtime equally as new stations are added.

These dynamic results show that PoliFi has a short reaction time, and can continuously enforce airtime usage policies as station activity changes. This is important for deployment in a real network with varying activity levels.

4.3 DASH Traffic Test

To showcase an example real-world use case that can be improved by airtime policy enforcement, we examine a DASH video streaming application. In this scenario, we add a station with poor signal quality to the network, representing a streaming device that is connected to the wireless network at a location where signal quality is poor. Moving the device is not an option, so other measures are necessary to improve the video quality. We stream the Big Buck Bunny [11] video using the dash.js [12] player running in the Chromium browser on the slow station. We determine that the maximum video bitrate the device can reliably achieve in this scenario (with no competing traffic) is 2 Mbps. However, when the other devices are active, the video bitrate drops to 1 Mbps because of contention.

Figure 5 shows the achieved video bitrate along with the data goodput of the video flow, while three other stations are simultaneously receiving bulk data. With no policy set, the video bitrate drops to 1 Mbps, as described above. However, when we prioritise the station (to half the available airtime in this case), the achieved bitrate stays at 2 Mbps throughout the 10-minute video. This shows how PoliFi can improve the performance of a specific real-world application.

5 Conclusion

We have presented PoliFi, a solution for enforcing airtime usage policies in WiFi networks. Our evaluation shows that PoliFi makes it possible to express a range of useful policies, including prioritisation of specific devices, and balancing or limiting of groups of devices. We have also shown how the policy enforcement can improve the performance of a real-world DASH video streaming application.

PoliFi can improve performance of WiFi networks with high airtime contention, and enables novel network usages such as network slicing. For this

reason we believe it to be an important addition to modern WiFi networks, which we are working to make widely available through inclusion in the upstream Linux WiFi stack.

References

[1] M. Heusse *et al.*, "Performance anomaly of 802.11 b," in *IEEE INFOCOM 2003. Twenty-Second Annual Joint Conference of the IEEE Computer and Communications Societies*, vol. 2. IEEE, 2003, pp. 836–843.

[2] T. Høiland-Jørgensen *et al.*, "Ending the anomaly: Achieving low latency and airtime fairness in wifi," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, 2017.

[3] X. Foukas *et al.*, "Network slicing in 5g: Survey and challenges," *IEEE Communications Magazine*, vol. 55, no. 5, pp. 94–100, 2017.

[4] J. Babiarz, K. Chan, and F. Baker, "Configuration Guidelines for DiffServ Service Classes," RFC 4594 (Informational), RFC Editor, Aug. 2006.

[5] T. Szigeti, J. Henry, and F. Baker, "Mapping Diffserv to IEEE 802.11," RFC 8325 (Proposed Standard), RFC Editor, Feb. 2018.

[6] M. Richart *et al.*, "Resource allocation for network slicing in WiFi access points," in *13th International Conference on Network and Service Management, CNSM, 2017*, 2017.

[7] K. Katsalis *et al.*, "Virtual 802.11 wireless networks with guaranteed throughout sharing," in *2015 IEEE Symposium on Computers and Communication (ISCC)*, Jul 2015.

[8] Y. Yiakoumis *et al.*, "Slicing home networks," in *Proceedings of the 2Nd ACM SIGCOMM Workshop on Home Networks*, ser. HomeNets '11. ACM, 2011.

[9] "Air Time Fairness (ATF) Phase1 and Phase 2 Deployment Guide," Cisco systems, 2015. https://www.cisco.com/c/en/us/td/docs/wireless/technology/mesh/8-2/b_Air_Time_Fairness_Phase1_and_Phase2_Deployment_Guide.html

[10] T. Høiland-Jørgensen *et al.*, "The Flow Queue CoDel Packet Scheduler and Active Queue Management Algorithm," RFC 8290 (Experimental), RFC Editor, Jan. 2018.

[11] "Big Buck Bunny," Blender Foundation, 2018. https://peach.blender.org/

[12] "dash.js reference DASH player," Dash Industry Forum, 2018. https://github.com/Dash-Industry-Forum/dash.js/wiki

Paper VII

Adapting TCP Small Queues for IEEE 802.11 Networks

Reprinted from

The 29th Annual IEEE International Symposium on Personal, Indoor and Mobile Radio Communications (IEEE PIMRC 2018), 9-12 September 2018, Bologna, Italy

"A problem never exists in isolation; it is surrounded by other problems in space and time."

Russell L. Ackoff

Adapting TCP Small Queues for IEEE 802.11 Networks

Carlo Augusto Grazia, Natale Patriciello, Toke Høiland-Jørgensen, Martin Klapez and Maurizio Casoni

Abstract

In recent years, the Linux kernel has adopted an algorithm called TCP Small Queues (TSQ) for reducing queueing latency by controlling buffering in the networking stack. This solution consists of a back-pressure mechanism that limits the number of TCP segments within the sender TCP/IP stack, waiting for packets to actually be transmitted onto the wire before enqueueing further segments. Unfortunately, TSQ prevents the frame aggregation mechanism in the IEEE 802.11n/ac standards from achieving its maximum aggregation, because not enough packets are available in the queue to build aggregates from, which severely limits achievable throughput over wireless links. This paper demonstrates this limitation of TSQ in wireless networks and proposes Controlled TSQ (CoTSQ), a solution that improves TSQ so that it controls the amount of data buffered while allowing the IEEE 802.11n/ac aggregation logic to fully exploit the available channel and achieve high throughput. Results on a real testbed show that CoTSQ leads to a doubling of throughput on 802.11n and up to an order of magnitude improvement in 802.11ac networks, with a negligible latency increase.

1 Introduction

The Linux networking stack has seen many improvements and much research activity in recent years, with the goals of reducing congestion and latency, and mitigating the bufferbloat phenomenon [1, 2]. This research has included the introduction of novel Active Queue Management (AQM) algorithms such as Codel [3] and PIE [4] as well as novel TCP congestion control algorithms like BBR [5]. Another essential solution, enabled by default in the Linux kernel since 2012, is TCP Small Queues (TSQ), a fine-grained backpressure mechanism that limits each TCP flow to 1 ms of data at the current rate in the TCP/IP stack; subsequent packets from the same flow are enqueued only

after the actual NIC transmission. However, not much analysis of TSQ is available in the scientific literature; in fact, only a few research articles have at most cited this algorithm [6–8] spending few words, or nothing at all, on the TSQ mechanism itself. To the best of our knowledge, only Guo et al. in [9] investigated TSQ by interacting with the algorithm and changing the threshold for experimental purposes. Unfortunately, this work is tailored for cellular networks only, as it focuses entirely on latency where the TSQ threshold is reduced in order to eliminate queueing as much as possible. The paper concluded that reducing the TSQ threshold has a negligible impact on the firmware queueing delay. In addition, the TSQ tuning provided in [9] cannot be applied to the new Linux kernel versions in which the TSQ policy has been hard-coded in the kernel and cannot be modified by the user. In wired networks, TSQ largely achieves its goal of latency reduction without affecting the maximum achievable throughput. However, the same does not hold for wireless networks, such as IEEE 802.11n/ac WLANs, and this problem has never been discussed in the literature. The contributions of this paper are the following: (i) we demonstrate the throughput degradation in IEEE 802.11n/ac WLANs caused by TSQ; (ii) we propose Controlled TSQ (CoTSQ), a solution to modify and control the TSQ behaviour through a Linux kernel patch and (iii) we evaluate our solution on a real testbed providing an extensive set of tests that show how CoTSQ doubles the achievable throughput in 802.11n networks and improves it by up to an order of magnitude in 802.11ac networks, in both cases with a negligible latency increase. The rest of the paper is organised as follows: Section 2 details the TSQ algorithm, while Section 3 presents our CoTSQ solution. Section 4 describes the testbed used to produce the results available in Section 5. Finally, Section 6 concludes the paper.

2 TCP Small Queues in a Nutshell

This section describes the TSQ mechanism, first implemented by Eric Dumazet and available since Linux kernel versions 3.6 [10]. To understand TSQ, we refer to Figure 1 that shows the path of TCP packets, from the TCP socket of the host through the different layers of the Linux networking stack: The TCP layer, the queueing layer and the device driver.

At the top of Figure 1 is the TCP layer. When an application transmits data through a TCP socket, it is queued in the socket buffer. The TCP control system determines when packets are dequeued from the socket buffer and moved into the lower levels of the network stack. This control system consists of the Congestion Control algorithm, the TSQ logic and the Pacing mechanism, which can each limit the flow of packets into lower layers. The Congestion Control algorithm is the core engine of the protocol, and each TCP variant (such as Cubic, BBR, Vegas, New Reno, etc.) define different behaviour for this congestion control module. The module implements the standard ACK logic of TCP, maintains the congestion window, etc. TSQ and Pacing have two different goals: the former has the role of limiting the amount of data of any given socket in the stack, while the latter smoothes

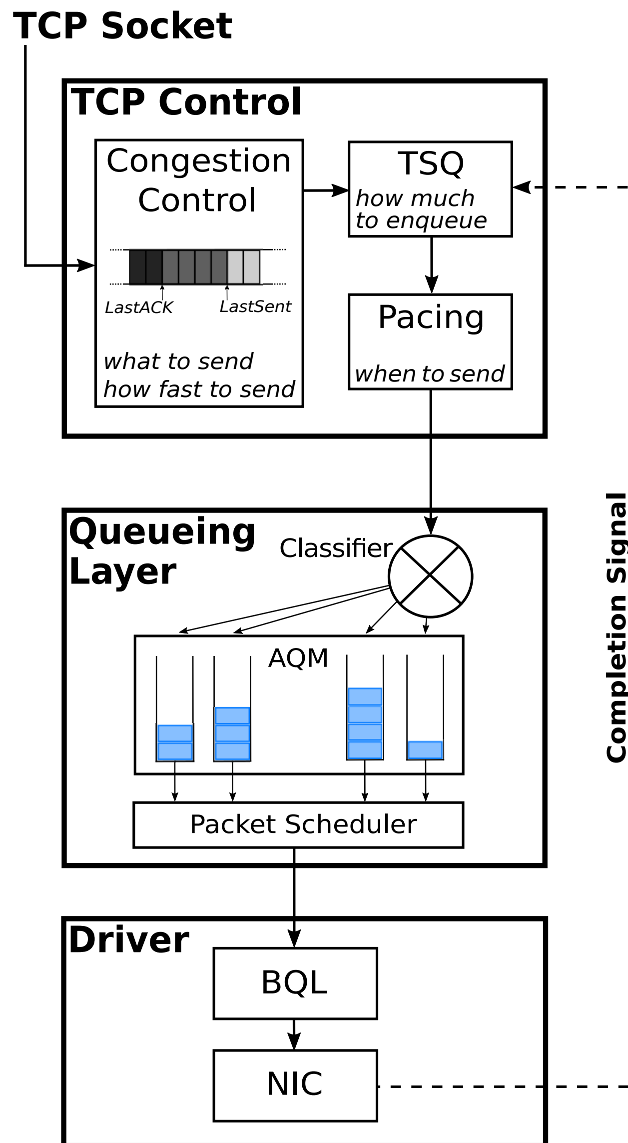


Figure 1: Linux TCP sender architecture.

out transmission of individual packets by spacing them equally according to the current estimated rate, thus preventing big bursts of packets put onto the medium at once. The pacing mechanism is optional and can be activated by the congestion control module or by the sending application, while TSQ and the selected congestion control is always active.

Inside the queueing layer, packets are managed according to the queueing discipline deployed. The queueing layer can implement various queueing mechanisms, ranging from a simple FIFO queue, to elaborate queueing schemes with per-flow queueing and fairness enforcement. In addition, AQM can be implemented in this layer, either as a simple mechanism on top of a FIFO queue, or as part of a more elaborate multi-queue system.

A lot of the scientific work seeking to address bufferbloat has focused on the queueing layer. The Linux kernel implements the queueing layer as a generic *queueing discipline* (qdisc) which is used for all networking interfaces. The current state of the art qdisc (which is used by default in many deployments of Linux) is FQ-Codel [11], which is a fairness queueing algorithm that uses the CoDel AQM to control each queue, and a Deficit Round Robin-based scheduling of the queues. However, for supported WiFi drivers, the kernel instead uses an integrated queueing system based on the FQ-CoDel algorithm, but tailored specifically for WiFi [12].

Once the queueing layer dequeues a packet, the latter moves to the last block in the figure which is the device driver controlling the Network Interface Card (NIC). Here there is a buffer that manages the packets waiting to be sent on the physical link, and the Linux kernel deploys solutions to mitigate bufferbloat also at this low level of the network stack. This solution is named Byte Queue Limits (BQL) and puts a cap on the amount of data that can be enqueued and waits to be delivered by the NIC [13]. The last block also contains the implementation of the IEEE 802.11n/ac aggregation logic, which is implemented either in the device driver or in the device firmware itself. Aggregation combines several packets into a single link-layer frame, reducing overhead and increasing throughput [14, 15].

As seen above, there are queues in several different layers of the network stack, and thus several places in which bufferbloat can occur, each requiring its own solution. TSQ is complementary to the other solutions, and is targeted specifically at applications on the local host that use TCP; by providing backpressure, these applications can better react to congestion signals for their traffic, and achieve lower application-level latency. TSQ controls dequeueing from the socket buffer, and so it limits the enqueued data of any TCP flow regardless of where the data is queued. When the limit is reached, new data of a flow can be enqueued only when previously enqueued packets from the same flow have been dequeued by the NIC, and the packet memory freed. This works by triggering a completion signal that notifies the TSQ logic when the driver frees the memory allocated for a data packet.

The amount of queue allowed by the TSQ logic is adjusted dynamically as a function of the TCP rate; the amount of data that can be enqueued for each flow corresponds to the amount of data that can be transmitted in 1 ms at the

current rate of the flow. This policy is fixed in the kernel, and the TSQ logic can be neither disabled nor tuned.

3 Controlled TSQ

To overcome the inflexible behavior of TSQ, we patched the kernel to expose the TSQ core parameters and make it possible to disable or tune the TSQ logic. We name this solution Controlled TSQ (CoTSQ) [16]. The CoTSQ logic relies on 3 parameters:

- `bytes` is the TSQ parameter that limits the amount of data, expressed in bytes, for each TCP flow. It is used to impose an upper bound on the queue;
- `ms` limits the amount of data in the stack as a function of the latency. It gives the dynamic threshold, autotuning the number of bytes to enqueue as a function of the current rate;
- `pkts` sets a *lower* bound on the number of packets queued for each TCP flow.

$$B = \min\left(\texttt{bytes}, \max\left(\texttt{ms}, \texttt{pkts}\right)\right) \tag{1}$$

These three parameters are combined into the queue limit `B` enforced by TSQ as given in Equation (1). The `ms` and `pkts` parameters are first converted to bytes, by multiplying the former by the current flow rate and the latter by the TCP Maximum Segment Size (MSS). We chose Equation (1) because it is easy to define `bytes` as upper bound (maximum amount of data to be enqueued), `pkts` as lower bound (minimum amount of packets to be enqueued) and `ms` as the core value. The latter gives the actual amount of data to enqueue as a function of the rate. We can express the standard TSQ behavior by setting `bytes`=128KB, `ms`=1 and `pkts`=0 in the CoTSQ parameters. Similarly, we also defined a way to completely disable TSQ by setting `bytes`=-1 (or any negative value) for testing purposes.

4 Testbed

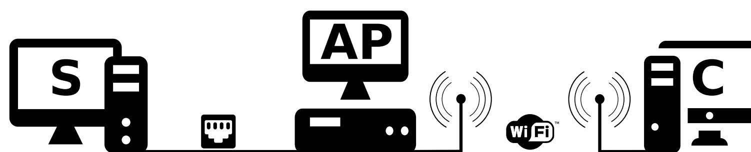


Figure 2: Physical testbed layout.

Table 1: Testbed setups

setup name	chipset	protocol
ath9k_htc	Atheros AR9271 (1x1 MIMO)	802.11n
ath9k	Atheros AR9580 (3x3 MIMO)	802.11n
ath10k	Atheros QCA9880v2 (3x3 MIMO)	802.11ac

Table 2: Parameters

parameter	value
Kernel version	4.14.14-1
TCP Congestion Control	Cubic
TSQ type	TSQ, NoTSQ and xTSQ
Queueing discipline	FQ_Codel
Tests	1-4 TCP Uploads, RRUL

To evaluate our solution we deployed the physical testbed depicted in Figure 2. It is composed of three nodes: a wired server `S`, a wireless client `C` and an access point `AP` that acts as a router.

Nodes characteristics. All the three testbed nodes are running the Arch Linux distribution, equipped with the latest (at the time of writing) Linux kernel version 4.14.14-1. The end nodes `C` and `S` use CUBIC as their TCP congestion control algorithm. We tested three different wireless devices, a USB dongle and two types of PCIe cards. For all the device types, one device is used to create the Access Point router on the `AP` node and one other is used to provide wireless connectivity to `C`. The three setups are labelled by the WiFi device driver names as ath9k_htc, ath9k, and ath10k, and are summarised in Table 1.

The ath9k_htc setup uses USB devices which contain Qualcomm Atheros Communications AR9271 chipsets. The ath9k setup uses PCIe devices containing Qualcomm Atheros Communications AR9580 chipsets with 3x3 MIMO antennas. These first two setups are used for testing the IEEE 802.11n protocol. The ath10k setup uses PCIe devices containing Qualcomm Atheros Communications QCA9880v2 chipsets with 3x3 MIMO antennas. This latter setup is used for testing the IEEE 802.11ac protocol. These three setups, due to the different drivers, have different queueing systems. The ath9k_htc driver uses the standard qdisc-based queueing layer. The qdisc is configured to use FQ-CoDel as the queueing discipline, which is the default in Arch Linux and it is also the best option for preventing bufferbloat [17]. The ath9k and ath10k drivers instead use the integrated WiFi-specific queueing layer mentioned previously. The experiments on the ath9k and ath10k setups were carried out at Karlstad University, in a setup similar to the one shown in Figure 2, except that tests were run between `AP` and `C`.

TSQ is the main variable parameter for our tests and can have different values:

- TSQ: the standard TSQ behavior that is obtained by setting `bytes` = 128KB, `ms` = 1 and `pkts` = 0;

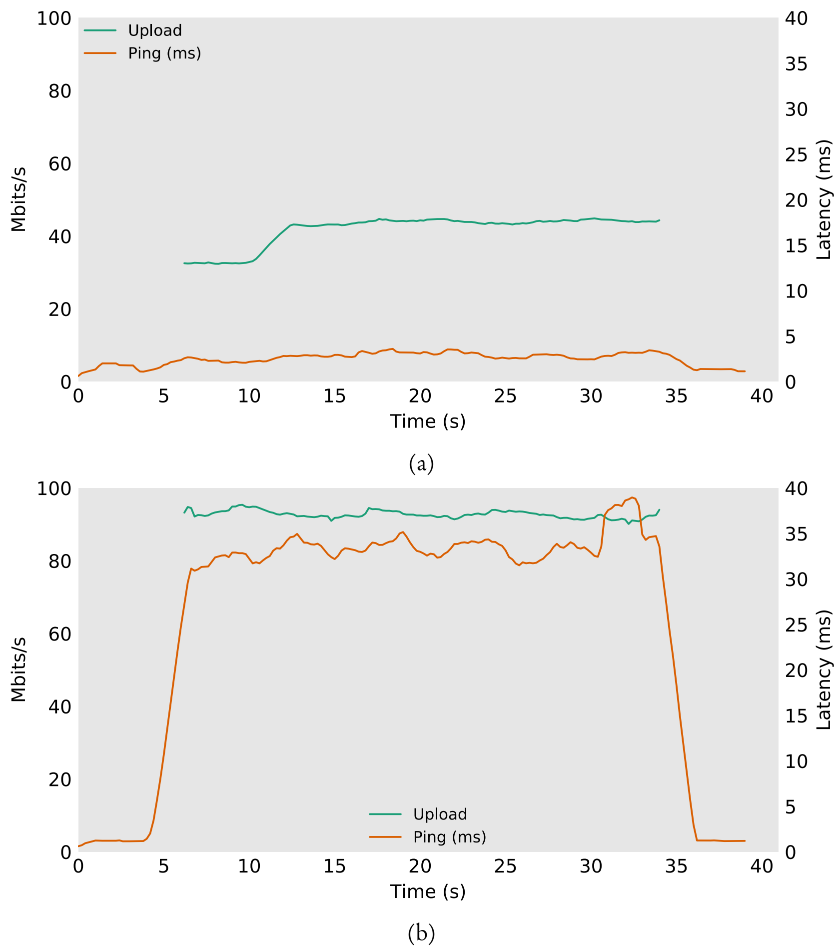


Figure 3: Goodput vs latency, 1 TCP upload on ath9k_htc TSQ (a, top) and ath9k NoTSQ (b, bottom).

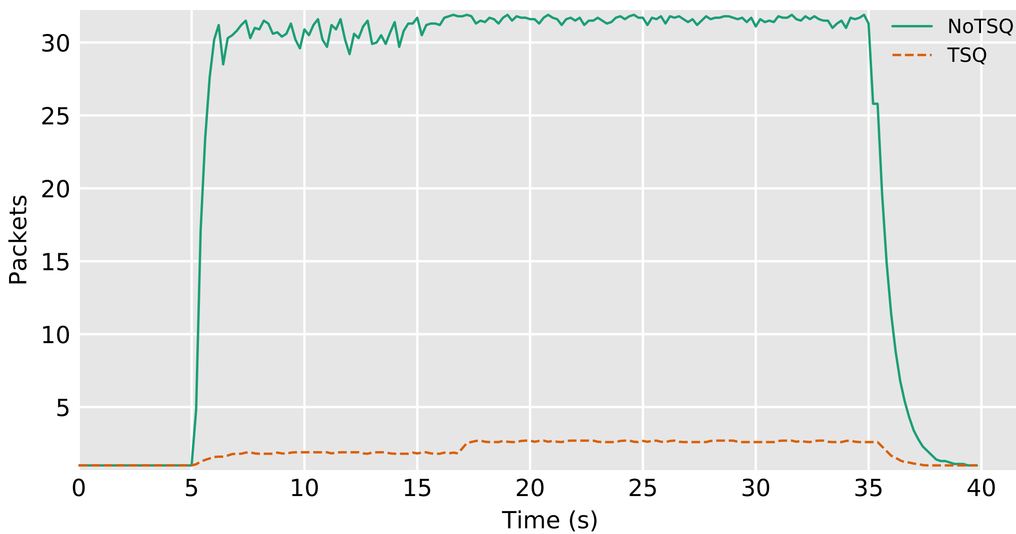


Figure 4: Frame aggregation over time (ath9k).

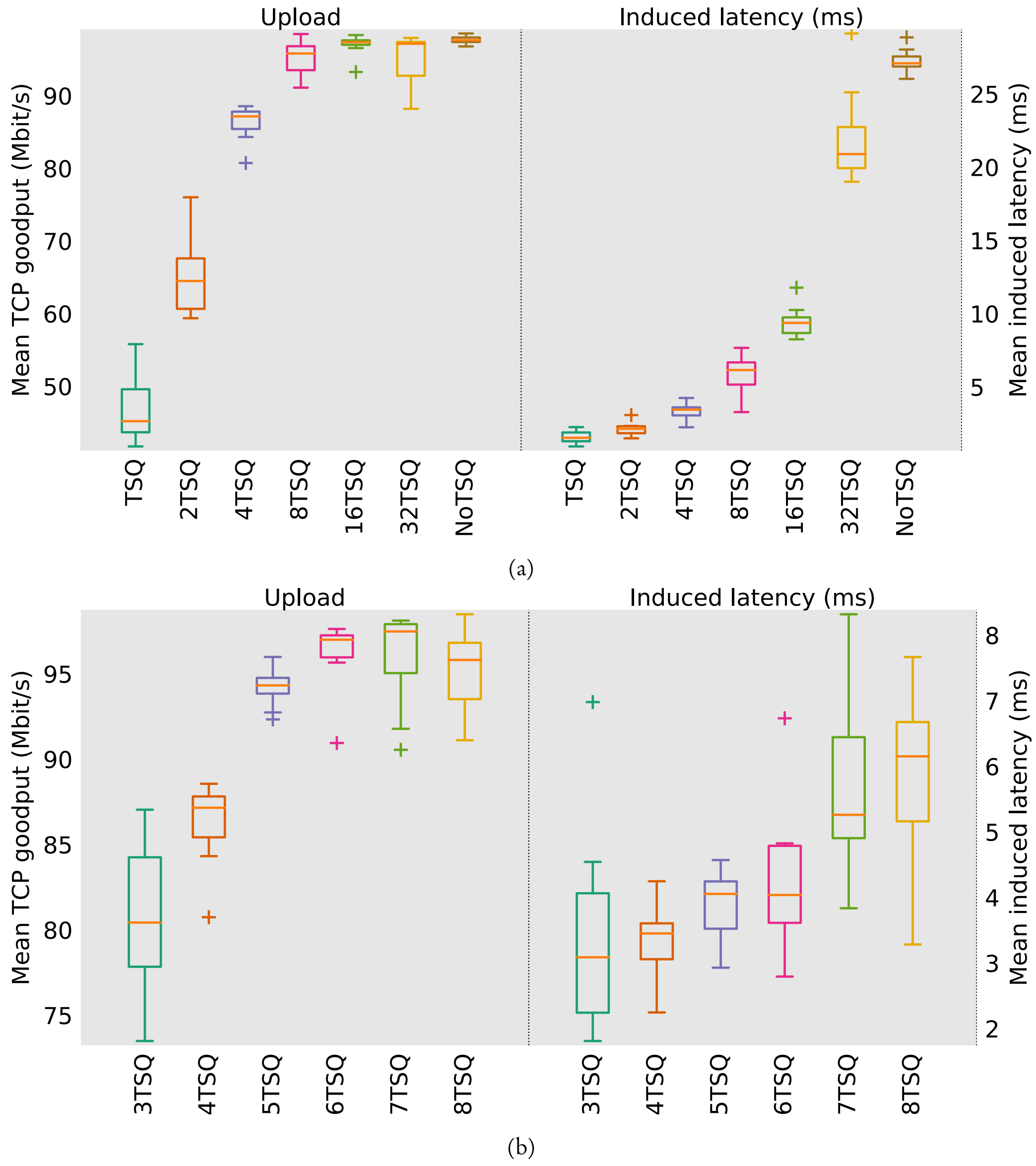


Figure 5: Goodput Vs Latency during a TCP flow upload on ath9k_htc. (a) Exponentially increasing `ms` values. (b) Linearly increasing `ms` values.

- NoTSQ: that completely disable TSQ logic and is obtained by setting `bytes=-1`;

- xTSQ: where `x` is the value, expressed in milliseconds, of the amount of data to enqueue for each TCP flow. It is obtained by setting `bytes=10MB`, `ms=x` and `pkts=1`. This expresses the tunable nature of the CoTSQ patch. We have set up a lower bound of one packet and an upper bound of 10MB of data, while the amount `x` rules the CoTSQ behavior by determining the amount of data to enqueue as a function of the TCP rate, autotuning the threshold like the original TSQ implementation. These bounds have been selected simply to make sure only the `ms` value controls the queue.

Test configuration. We performed our tests using Flent [18], a flexible network tester able to configure different kinds of traffic and collect the results. We performed tests with a variable number of flows: starting from a single flow in upload, then moving to a more congested network with up to four flows in upload. Each test consists of one minute of actual TCP flow; a 5-seconds period of only latency measurement is included before and after each test to better show the impact of TCP on the queueing delay. We also run a test named Real-time Response Under Load (RRUL) designed by the bufferbloat community for analyzing network performance under heavy workloads; it consists in 8 TCP flows, 4 in download and 4 in upload, that compete with ICMP and UDP traffic. The RRUL test has been included in order to measure the impact on bidirectional traffic. All the testbed characteristics are summarised in Table 2.

5 Results

In this section, we analyse the results obtained by running the aforementioned tests on the physical testbed. Only a subset of the results are presented here, but the full dataset is available together with the patch itself, and the test scripts, in [16]. Where not explicitly stated otherwise, the results are from the ath9k_htc setup.

The frame aggregation problem. We first present Figure 3 that encapsulates, in a sense, the motivation of this paper. The Figure shows the goodput achieved by a single TCP flow in upload (from the client to the server) competing with a ping flow. Figure 3a, shows the default configuration with TSQ enabled, while Figure 3b shows the result with the TSQ logic completely disabled (labeled NoTSQ).

This Figure has several interesting features:

- The default configuration with TSQ enabled gives a very low latency that oscillates between 1 and 4 ms. At the same time, the goodput is firmly bounded at 44Mbit/s;

- The patched configuration with NoTSQ (TSQ disabled) results clearly in a higher goodput that oscillates between 90 and 95 Mbit/s, which is

the channel bandwidth limit with the equipped 802.11n hardware of the ath9k_htc setup. At the same time, the latency is increased to around 35ms.

The goodput/latency tradeoff between TSQ and NoTSQ is evident from this test, as well as the TSQ limitation in channel exploitation over an 802.11n environment. The reason why TSQ does not provide optimal goodput lies in the achieved frame aggregation, which is shown in Figure 4; the presence of a limited number of packets queued in the driver limits the size of the aggregates that can be built, which severely limits the achievable throughput. On the other hand, disabling TSQ (with the NoTSQ test) lets the system reach the maximum goodput, better utilising the channel, but inducing a latency that reaches 35ms, ten times higher than with TSQ enabled. The goodput limit, in this case, strongly depends on the WLAN cards hardware.

The natural question at this point is: is it possible to fully utilise the channel, reaching high goodput through frame aggregation while maintaining a low latency?

The CoTSQ solution. To explore the tradeoff between latency and throughput, we performed additional experiments while varying the `ms` parameter. Figure 5a shows the results of exponentially increasing the parameter value, as well as the effect of completely disabled TSQ. The figure shows that while the latency keeps increasing with the allowed queue space, throughput only increases up to the 8TSQ configuration. We then proceeded to a fine-grained evaluation of xTSQ switching to a linear increment of values in the range between three and eight `ms`. Figure 5b shows that the optimum tradeoff is found at 6 ms, which we will refer to as CoTSQ for the rest of this paper. Here, throughput reaches the maximum value while maintaining a latency only 2ms higher than the standard TSQ.

To examine the impact of CoTSQ in a scenario with more congestion, we perform an additional evaluation in a scenario where 4 simultaneous TCP uploads are active. Figure 6 shows the combined goodput of the four flows, and the latency measured while they were active. The first thing to notice is that the trend is maintained, CoTSQ and NoTSQ almost double the goodput also in a more congested environment with four transmitting flows. The second important thing to consider is that the latency induced by the CoTSQ solution is not increased by the presence of more TCP flows, and remains at 6 ms.

What happens in the presence of bidirectional traffic is shown by the RRUL test. In this case, TSQ only impacts the upstream traffic. This is because the client TSQ logic does not impact sending of ACKs from `C`, and because the bottleneck is at the `AP`, so there is no queue at `S`, making TSQ irrelevant. Figure 7 shows the results of the RRUL test run, and includes the goodput value of both download and upload streams, together with the induced latency. A very first thing to notice is the difference between the goodput in the download and upload directions. When standard TSQ is in place, the download is unfairly favoured; the difference is a factor of 3 with more than 75 Mbit/s for the download and less than 25 Mbit/s for the upload.

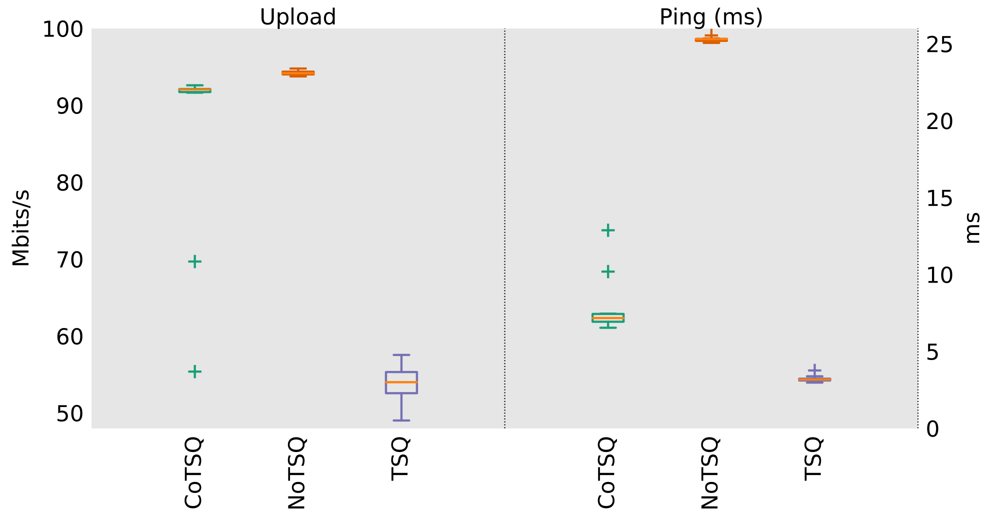


Figure 6: Goodput vs Latency, 4 TCP flows in upload.

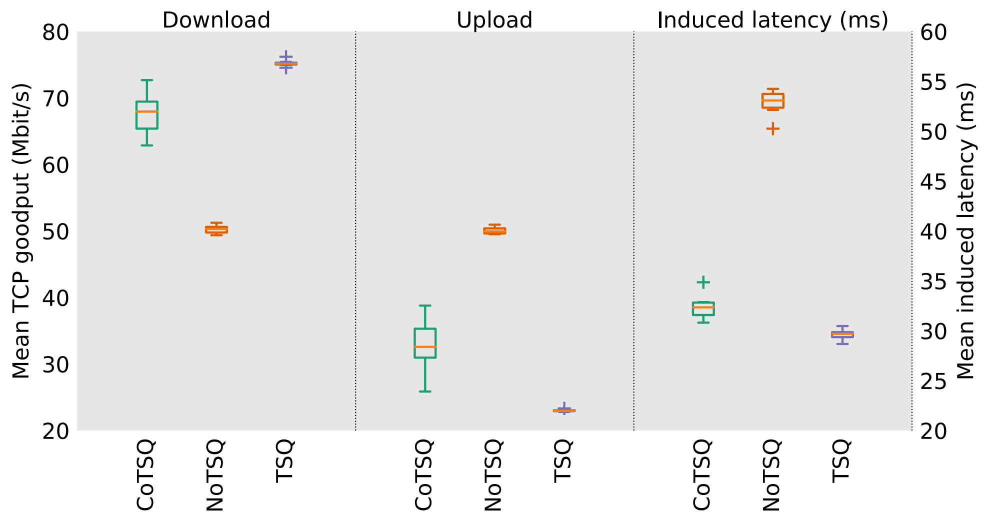


Figure 7: Goodput vs Latency, RRUL test.

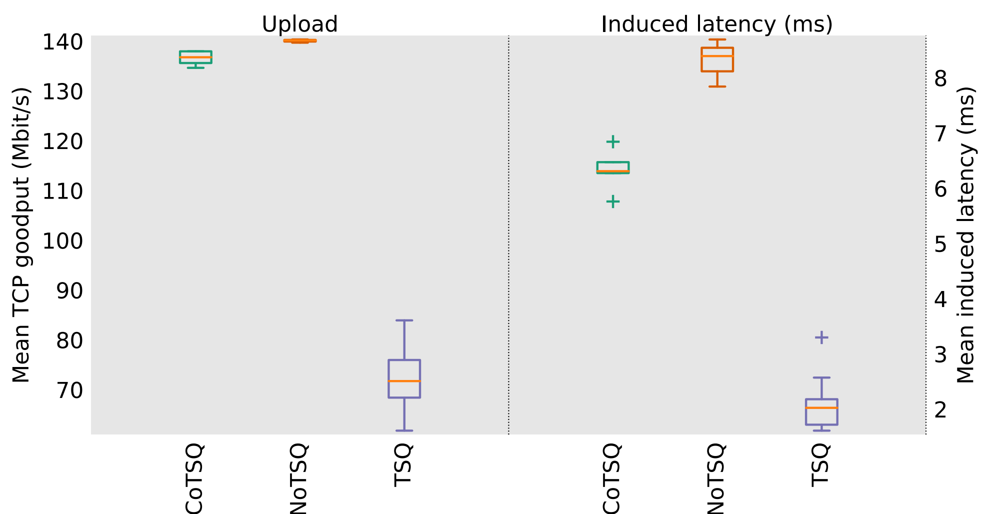


Figure 8: Goodput vs Latency, 1 TCP upload: ath9k setup.

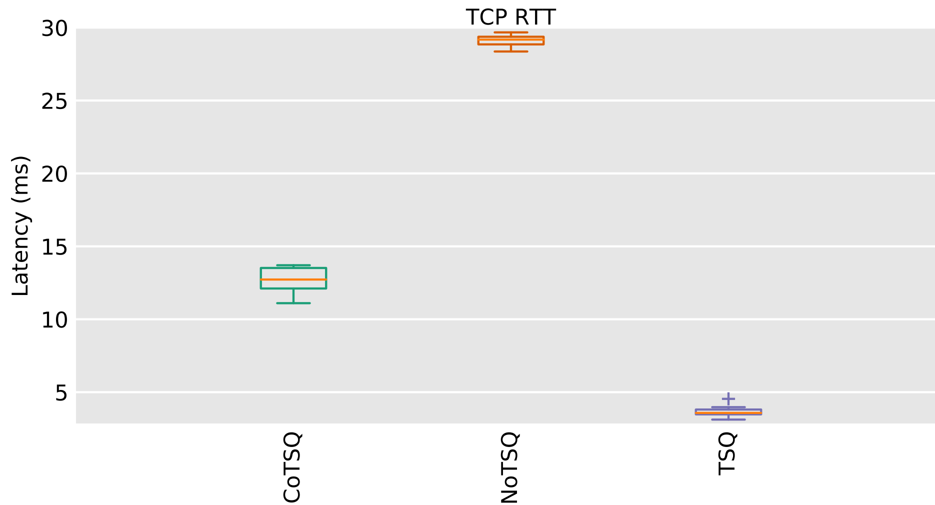


Figure 9: TCP RTT, 1 TCP upload: ath9k setup.

The reason for such unfairness is that while the upload traffic is limited by TSQ, the download traffic is not, and so has no problem reaching its maximum achievable throughput. CoTSQ reduces the difference between upload and download, while keeping latency close to the value achieved by TSQ. While NoTSQ achieves perfect fairness between upload and download, this comes at the cost of almost twice the latency compared to TSQ and CoTSQ.

To show the impact of CoTSQ with a different queueing structure, Figure 8 shows the goodput of a single TCP upload together with the induced latency on the ath9k configuration. The first difference to notice is that the ath9k hardware achieves a higher maximum throughput due to its 3x3 MIMO setup. However, the same relative impact of TSQ remains, with CoTSQ doubling the achievable throughput relative to TSQ. As before, the latency increases to 6 ms with CoTSQ. The second difference to notice with Figure 6, is that the mean induced latency on NoTSQ is reduced from almost 25 ms in the ath9k_htc setup, to less than 10 ms in the ath9k setup. This is because the WiFi-specific queueing structure used by the ath9k driver more effectively isolates the latency measurement (ping) flow from the queue induced by the TCP flow. To show the benefit of CoTSQ compared to NoTSQ in this setup, we also show the RTT of the TCP flow itself during the same test, in Figure 9. From this it is clear that CoTSQ helps the TCP sender to maintain a lower RTT while still reaching almost optimal goodput; the difference between the CoTSQ RTT and the NoTSQ RTT is nearly 20 ms.

802.11ac. Figure 10a shows the goodput of a single TCP upload together with the induced latency for the ath10k setup. This clearly shows the increased maximum bandwidth of 802.11ac compared to 802.11n. The remarkable thing to notice is that the goodput improvement from TSQ to CoTSQ in this setup is even larger, showing an order of magnitude improvement. Figure 10b shows the repeat of the experiment exploring different parameter values for TSQ. From this, it is clear the optimal value of `ms` is still 6 ms of data to enqueue. However, increasing the `ms` beyond this does not lead to a stable maximum

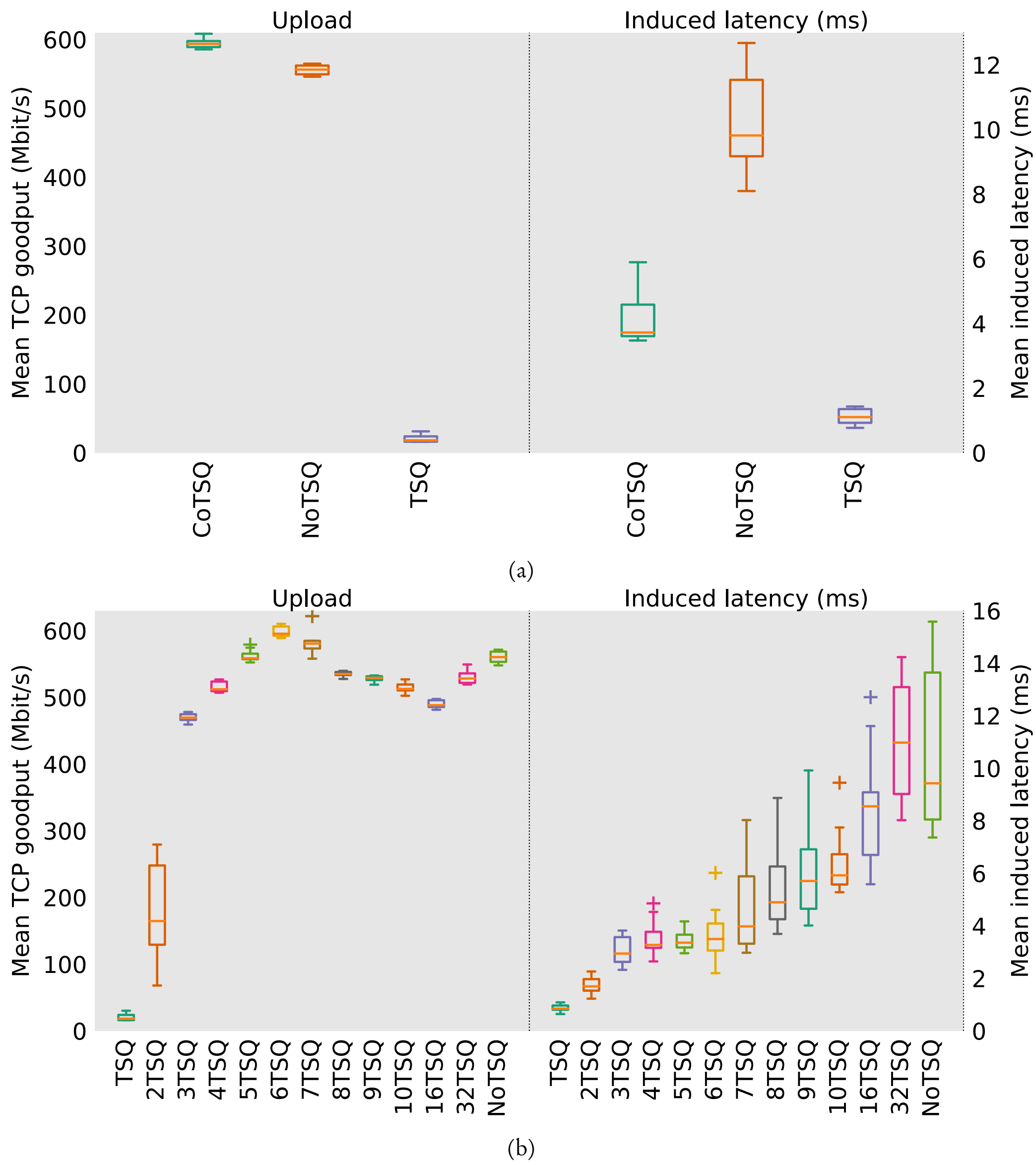


Figure 10: Goodput Vs Latency achieved during 1 TCP upload: ath10k setup. (a) CoTSQ, NoTSQ and TSQ comparison. (b) Different `ms` values.

goodput. We attribute this to interactions between queueing in the driver and queueing in the firmware, and plan to investigate this further in the future.

To conclude this section, we have demonstrated with CoTSQ that it is possible to achieve high TCP goodput over wireless 802.11 networks with a negligible latency induction. Based on our results, we have submitted a patch to the Linux kernel which modifies the TSQ parameters for all TCP flows going out over wireless interfaces. The patch has been accepted for inclusion and will be part of the upcoming version 4.17 of Linux.

6 Conclusions

We have designed and implemented CoTSQ, a solution that modifies the TSQ logic of the Linux kernel to allow each flow to enqueue enough data to avoid breaking the frame aggregation logic of wireless network interface cards. We have tested our solution in both 802.11n and 802.11ac networks, with different levels of traffic congestion. The results show that TSQ prevents full channel utilisation in wireless networks due to the inability to aggregate packets. Our CoTSQ solution doubles the achieves goodput in 802.11n networks, and improves it by an order of magnitude in 802.11ac networks, in both cases with a negligible increase in latency. Future work will investigate the benefit of CoTSQ with a wider range of different wireless drivers as well as with different TCP congestion control algorithms.

References

[1] J. Gettys and K. Nichols, "Bufferbloat: Dark Buffers in the Internet," *Queue*, vol. 9, no. 11, p. 40, 2011.

[2] Y. Gong *et al.*, "Fighting the Bufferbloat," in *2013 IEEE Conference on Computer Communications Workshops*, April 2013, pp. 411–416.

[3] K. Nichols and V. Jacobson, "Controlling Queue Delay," *Communications of the ACM*, vol. 55, no. 7, pp. 42–50, 2012.

[4] R. Pan *et al.*, "PIE: A lightweight control scheme to address the Bufferbloat problem," in *2013 IEEE 14th International Conference on High Performance Switching and Routing (HPSR)*. IEEE, 2013, pp. 148–155.

[5] N. Cardwell *et al.*, "BBR: Congestion-based Congestion Control," *Communications of the ACM*, vol. 60, no. 2, pp. 58–66, 2017.

[6] A. Saeed *et al.*, "Carousel: Scalable traffic shaping at end hosts," in *SIGCOMM 2017 - Proceedings of the 2017 Conference of the ACM Special Interest Group on Data Communication*, 2017, pp. 404–417.

[7] B. Briscoe *et al.*, "Reducing internet latency: A survey of techniques and their merits," *IEEE Communications Surveys and Tutorials*, vol. 18, no. 3, pp. 2149–2196, 2016.

[8] R. Lübben *et al.*, "On characteristic features of the application level delay distribution of TCP congestion avoidance," in *2016 IEEE International Conference on Communications, ICC 2016*, 2016.

[9] Y. Guo *et al.*, "Understanding on-device Bufferbloat for cellular upload," in *Proceedings of the 2016 Internet Measurement Conference*, 2016, pp. 303–317.

[10] J. Corbet, "TCP Small Queues," July 2012. https://lwn.net/Articles/507065/

[11] T. Høiland-Jørgensen *et al.*, "The Flow Queue CoDel Packet Scheduler and Active Queue Management Algorithm," RFC 8290 (Experimental), RFC Editor, Jan. 2018.

[12] T. Høiland-Jørgensen *et al.*, "Ending the anomaly: Achieving low latency and airtime fairness in wifi," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, 2017.

[13] J. Corbet, "Network transmit queue limits," LWN Article, August 2011. https://lwn.net/Articles/454390/

[14] J. Saldana, J. Ruiz-Mas, and J. Almodovar, "Frame aggregation in central controlled 802.11 WLANs: The latency versus throughput tradeoff," *IEEE Communications Letters*, vol. 21, no. 11, pp. 2500–2503, 2017.

[15] Y. Kim *et al.*, "Throughput enhancement of IEEE 802.11 WLAN via frame aggregation," in *IEEE Vehicular Technology Conference*, vol. 60, no. 4, 2004, pp. 3030–3034.

[16] "CoTSQ: Linux Kernel patch and source scripts," March 2018. http://netlab.ing.unimo.it/sw/cotsq.zip

[17] T. Høiland-Jørgensen, P. Hurtig, and A. Brunstrom, "The Good, the Bad and the WiFi: Modern AQMs in a residential setting," *Computer Networks*, vol. 89, pp. 90 – 106, 2015.

[18] T. Høiland-Jørgensen *et al.*, "Flent: The FLExible Network Tester," *ValueTools 2017*, 2017.

Paper VIII

The eXpress Data Path

Fast Programmable Packet Processing in the Operating System Kernel

Reprinted from

CoNEXT ’18: International Conference on emerging Networking EXperiments and Technologies, December 4–7, 2018, Heraklion, Greece

"I feel a need… A need for speed."

Pete "Maverick" Mitchell, Top Gun

The eXpress Data Path

Fast Programmable Packet Processing in the Operating System Kernel

Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern and David Miller

toke.hoiland-jorgensen@kau.se, brouer@redhat.com, daniel@cilium.io, john@cilium.io, tom@herbertland.com, dsahern@gmail.com, davem@redhat.com

Abstract

Programmable packet processing is increasingly implemented using kernel bypass techniques, where a userspace application takes complete control of the networking hardware to avoid expensive context switches between kernel and userspace. However, as the operating system is bypassed, so are its application isolation and security mechanisms; and well-tested configuration, deployment and management tools cease to function.

To overcome this limitation, we present the design of a novel approach to programmable packet processing, called the eXpress Data Path (XDP). In XDP, the operating system kernel itself provides a safe execution environment for custom packet processing applications, executed in device driver context. XDP is part of the mainline Linux kernel and provides a fully integrated solution working in concert with the kernel's networking stack. Applications are written in higher level languages such as C and compiled into custom byte code which the kernel statically analyses for safety, and translates into native instructions.

We show that XDP achieves single-core packet processing performance as high as 24 million packets per second, and illustrate the flexibility of the programming model through three example use cases: layer-3 routing, inline DDoS protection and layer-4 load balancing.

1 Introduction

High-performance packet processing in software requires very tight bounds on the time spent processing each packet. Network stacks in general purpose

operating systems are typically optimised for flexibility, which means they perform too many operations per packet to be able to keep up with these high packet rates. This has led to the increased popularity of special-purpose toolkits for software packet processing, such as the Data Plane Development Kit (DPDK) [1]. Such toolkits generally bypass the operating system completely, instead passing control of the network hardware directly to the network application and dedicating one, or several, CPU cores exclusively to packet processing.

The kernel bypass approach can significantly improve performance, but has the drawback that it is more difficult to integrate with the existing system, and applications have to re-implement functionality otherwise provided by the operating system network stack, such as routing tables and higher level protocols. In the worst case, this leads to a scenario where packet processing applications operate in a completely separate environment, where familiar tooling and deployment mechanisms supplied by the operating system cannot be used because of the need for direct hardware access. This results in increased system complexity and blurs security boundaries otherwise enforced by the operating system kernel. The latter is in particular problematic as infrastructure moves towards container-based workloads coupled with orchestration systems such as Docker or Kubernetes, where the kernel plays a dominant role in resource abstraction and isolation.

As an alternative to the kernel bypass design, we present a system that adds programmability directly in the operating system networking stack in a co-operative way. This makes it possible to perform high-speed packet processing that integrates seamlessly with existing systems, while selectively leveraging functionality in the operating system. This framework, called the eXpress Data Path (XDP), works by defining a limited execution environment in the form of a virtual machine running *eBPF* code, an extended version of original BSD Packet Filter (BPF) [2] byte code format. This environment executes custom programs directly in kernel context, before the kernel itself touches the packet data, which enables custom processing (including redirection) at the earliest possible point after a packet is received from the hardware. The kernel ensures the safety of the custom programs by statically verifying them at load time; and programs are dynamically compiled into native machine instructions to ensure high performance.

XDP has been gradually integrated into the Linux kernel over several releases, but no complete architectural description of the system as a whole has been presented before. In this work we present a high-level design description of XDP and its capabilities, and how it integrates with the rest of the Linux kernel. Our performance evaluation shows raw packet processing performance of up to 24 million packets per second per CPU core. While this does not quite match the highest achievable performance in a DPDK-based application on the same hardware, we argue that the XDP system makes up for this by offering several compelling advantages over DPDK and other kernel bypass solutions. Specifically, XDP:

- Integrates cooperatively with the regular networking stack, retaining full control of the hardware in the kernel. This retains the kernel security boundary, and requires no changes to network configuration and management tools. In addition, any network adapter with a Linux driver can be supported by XDP; no special hardware features are needed, and existing drivers only need to be modified to add the XDP execution hooks.
- Makes it possible to selectively utilise kernel network stack features such as the routing table and TCP stack, keeping the same configuration interface while accelerating critical performance paths.
- Guarantees stability of both the eBPF instruction set and the programming interface (API) exposed along with it.
- Does not require expensive packet re-injection from user space into kernel space when interacting with workloads based on the normal socket layer.
- Is transparent to applications running on the host, enabling new deployment scenarios such as inline protection against denial of service attacks on servers.
- Can be dynamically re-programmed without any service interruption, which means that features can be added on the fly or removed completely when they are not needed without interruption of network traffic, and that processing can react dynamically to conditions in other parts of the system.
- Does not require dedicating full CPU cores to packet processing, which means lower traffic levels translate directly into lower CPU usage. This has important efficiency and power saving implications.

In the rest of this paper we present the design of XDP and our performance analysis. This is structured as follows: Section 2 first outlines related work. Section 3 then presents the design of the XDP system and Section 4 presents our evaluation of its raw packet processing performance. Section 5 supplements this with examples of real-world use cases that can be implemented with XDP. Finally, Section 6 discusses future directions of XDP, and Section 7 concludes.

2 Related work

XDP is certainly not the first system enabling programmable packet processing. Rather, this field has gained momentum over the last several years, and continues to do so. Several frameworks have been presented to enable this kind of programmability, and they have enabled many novel applications. Examples of such applications include those performing single functions, such as switching [3], routing [4], named-based forwarding [5], classification [6],

caching [7] or traffic generation [8]. They also include more general solutions which are highly customisable and can operate on packets from a variety of sources [9–14].

To achieve high packet processing performance on Common Off The Shelf (COTS) hardware, it is necessary to remove any bottlenecks between the networking interface card (NIC) and the program performing the packet processing. Since one of the main sources of performance bottlenecks is the interface between the operating system kernel and the userspace applications running on top of it (because of the high overhead of a system call and complexity of the underlying feature-rich and generic stack), low-level packet processing frameworks have to manage this overhead in one way or another. The existing frameworks, which have enabled the applications mentioned above, take several approaches to ensuring high performance; and XDP builds on techniques from several of them. In the following we give a brief overview of the similarities and differences between XDP and the most commonly used existing frameworks.

The DataPlane Development Kit (DPDK) [1] is probably the most widely used framework for high-speed packet processing. It started out as an Intel-specific hardware support package, but has since seen a wide uptake under the stewardship of the Linux Foundation. DPDK is a so-called *kernel bypass* framework, which moves the control of the networking hardware out of the kernel into the networking application, completely removing the overhead of the kernel-userspace boundary. Other examples of this approach include the PF_RING ZC module [15] and the hardware-specific Solarflare Open-Onload [16]. Kernel bypass offers the highest performance of the existing frameworks [17]; however, as mentioned in the introduction, it has significant management, maintenance and security drawbacks.

XDP takes an approach that is the opposite of kernel bypass: Instead of moving control of the networking hardware *out* of the kernel, the performance-sensitive packet processing operations are moved *into* the kernel, and executed before the operating system networking stack begins its processing. This retains the advantage of removing the kernel-userspace boundary between networking hardware and packet processing code, while keeping the kernel in control of the hardware, thus preserving the management interface and the security guarantees offered by the operating system. The key innovation that enables this is the use of a virtual execution environment that verifies that loaded programs will not harm or crash the kernel.

Prior to the introduction of XDP, implementing packet processing functionality as a kernel module has been a high-cost approach, since mistakes can crash the whole system, and internal kernel APIs are subject to frequent change. For this reason, it is not surprising that few systems have taken this approach. Of those that have, the most prominent examples are the Open vSwitch [14] virtual switch and the Click [12] and Contrail [18] virtual router frameworks, which are all highly configurable systems with a wide scope, allowing them to amortise the cost over a wide variety of uses. XDP significantly lowers the cost for applications of moving processing into the kernel,

by providing a safe execution environment, and by being supported by the kernel community, thus offering the same API stability guarantee as every other interface the kernel exposes to userspace. In addition, XDP programs can completely bypass the networking stack, which offers higher performance than a traditional kernel module that needs to hook into the existing stack.

While XDP allows packet processing to move into the operating system for maximum performance, it also allows the programs loaded into the kernel to selectively redirect packets to a special user-space socket type, which bypasses the normal networking stack, and can even operate in a zero-copy mode to further lower the overhead. This operating mode is quite similar to the approach used by frameworks such as Netmap [19] and PF_RING [20], which offer high packet processing performance by lowering the overhead of transporting packet data from the network device to a userspace application, without bypassing the kernel completely. The Packet I/O engine that is part of PacketShader [4] is another example of this approach, and it has some similarities with special-purpose operating systems such as Arrakis [21] and ClickOS [22].

Finally, programmable hardware devices are another way to achieve high-performance packet processing. One example is the NetFPGA [23], which exposes an API that makes it possible to run arbitrary packet processing tasks on the FPGA-based dedicated hardware. The P4 language [24] seeks to extend this programmability to a wider variety of packet processing hardware (including, incidently, an XDP backend [25]). In a sense, XDP can be thought of as a “software offload”, where performance-sensitive processing is offloaded to increase performance, while applications otherwise interact with the regular networking stack. In addition, XDP programs that don’t need to access kernel helper functions can be offloaded entirely to supported networking hardware (currently supported with Netronome smart-NICs [26]).

In summary, XDP represents an approach to high-performance packet processing that, while it builds on previous approaches, offers a new tradeoff between performance, integration into the system and general flexibility. The next section explains in more detail how XDP achieves this.

3 The design of XDP

The driving rationale behind the design of XDP has been to allow high-performance packet processing that can integrate cooperatively with the operating system kernel, while ensuring the safety and integrity of the rest of the system. This deep integration with the kernel obviously imposes some design constraints, and the components of XDP have been gradually introduced into the Linux kernel over a number of releases, during which the design has evolved through continuous feedback and testing from the community.

Unfortunately, recounting the process and lessons learned is not possible in the scope of this paper. Instead, this section describes the complete system, by explaining how the major components of XDP work, and how they fit together to create the full system. This is illustrated by Figure 1, which shows

a diagram of how XDP integrates into the Linux kernel, and Figure 2, which shows the execution flow of a typical XDP program. There are four major components of the XDP system:

- **The XDP driver hook** is the main entry point for an XDP program, and is executed when a packet is received from the hardware.
- **The eBPF virtual machine** executes the byte code of the XDP program, and just-in-time-compiles it for increased performance.
- **BPF maps** are key/value stores that serve as the primary communication channel to the rest of the system.
- **The eBPF verifier** statically verifies programs before they are loaded to make sure they do not crash or corrupt the running kernel.

3.1 The XDP Driver Hook

An XDP program is run by a hook in the network device driver each time a packet arrives. The infrastructure to execute the program is contained in the kernel as a library function, which means that the program is executed directly in the device driver, without context switching to userspace. As shown in Figure 1, the program is executed at the earliest possible moment after a packet is received from the hardware, before the kernel allocates its per-packet *sk_buff* data structure or performs any parsing of the packet.

Figure 2 shows the various processing steps typically performed by an XDP program. The program starts its execution with access to a context object. This object contains pointers to the raw packet data, along with metadata fields describing which interface and receive queue the packet was received on.

The program typically begins by parsing packet data, and can pass control to a different XDP program through tail calls, thus splitting processing into logical sub-units (based on, say, IP header version).

After parsing the packet data, the XDP program can use the context object to read metadata fields associated with the packet, describing the interface and receive queue the packet came from. The context object also gives access to a special memory area, located adjacent in memory to the packet data. The XDP program can use this memory to attach its own metadata to the packet, which will be carried with it as it traverses the system.

In addition to the per-packet metadata, an XDP program can define and access its own persistent data structures (through BPF maps, described in Section 3.3 below), and it can access kernel facilities through various helper functions. Maps allow the program to communicate with the rest of the system, and the helpers allow it to selectively make use of existing kernel functionality (such as the routing table), without having to go through the full kernel networking stack. New helper functions are actively added by the kernel development community in response to requests from the community,

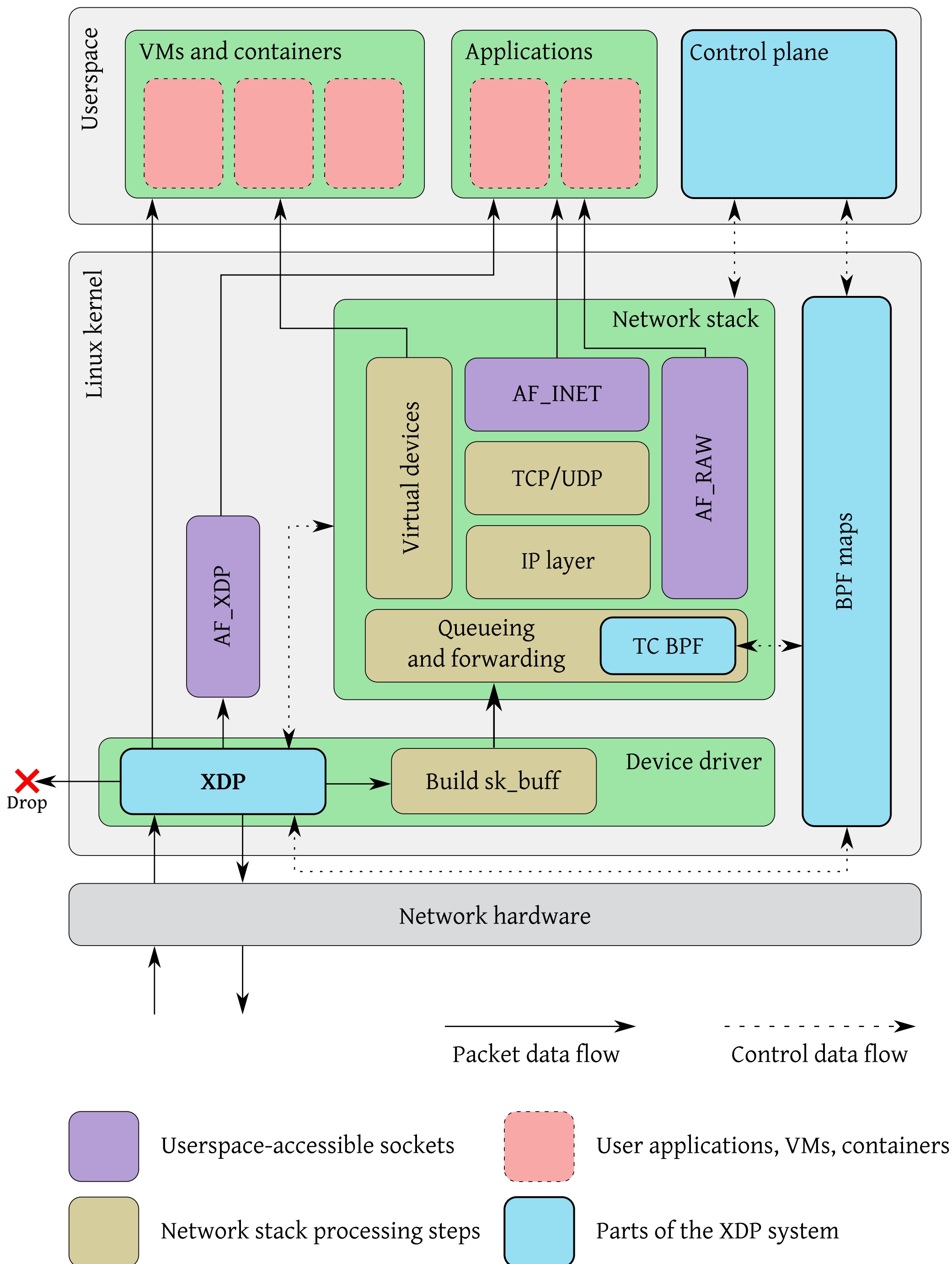


Figure 1: XDP's integration with the Linux network stack. On packet arrival, before touching the packet data, the device driver executes an eBPF program in the main XDP hook. This program can choose to drop packets; to send them back out the same interface it was received on; to redirect them, either to another interface (including vNICs of virtual machines) or to userspace through special AF_XDP sockets; or to allow them to proceed to the regular networking stack, where a separate TC BPF hook can perform further processing before packets are queued for transmission. The different eBPF programs can communicate with each other and with userspace through the use of BPF maps. To simplify the diagram, only the ingress path is shown.

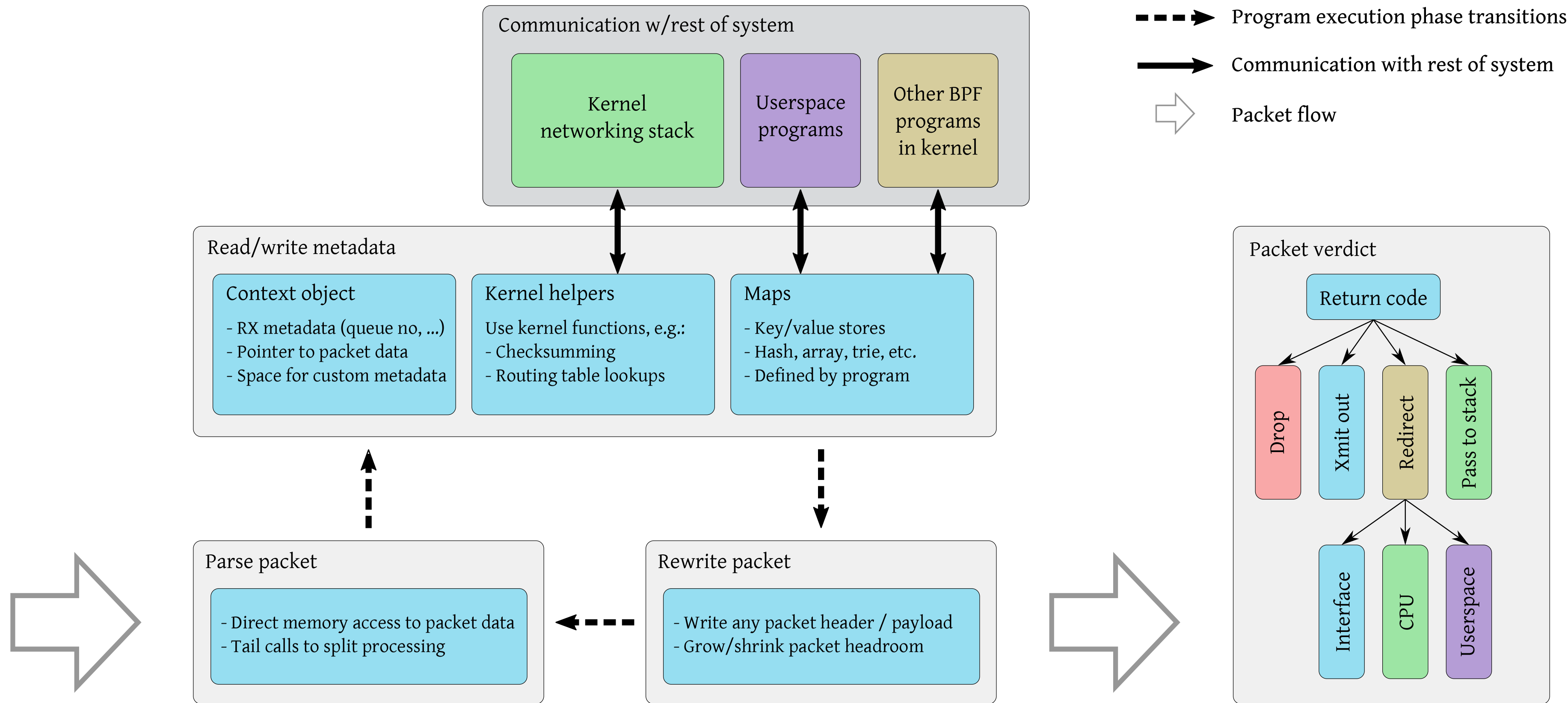


Figure 2: Execution flow of a typical XDP program. When a packet arrives, the program starts by parsing packet headers to extract the information it will react on. It then reads or updates metadata from one of several sources. Finally, a packet can be rewritten and a final verdict for the packet is determined. The program can alternate between packet parsing, metadata lookup and rewriting, all of which are optional. The final verdict is given in the form of a program return code.

thus continuously expanding the functionality that XDP programs can make use of.

Finally, the program can write any part of the packet data, including expanding or shrinking the packet buffer to add or remove headers. This allows it to perform encapsulation or decapsulation, as well as, for instance, rewrite address fields for forwarding. Various kernel helper functions are available to assist with things like checksum calculation for a modified packet.

These three steps (reading, metadata processing, and writing packet data) correspond to the light grey boxes on the left side of Figure 2. Since XDP programs can contain arbitrary instructions, the different steps can alternate and repeat in arbitrary ways. However, to achieve high performance, it is often necessary to structure the execution order as described here.

At the end of processing, the XDP program issues a final verdict for the packet. This is done by setting one of the four available return codes, shown on the right-hand side of Figure 2. There are three simple return codes (with no parameters), which can drop the packet, immediately re-transmit it out the same network interface, or allow it to be processed by the kernel networking stack. The fourth return code allows the XDP program to *redirect* the packet, offering additional control over its further processing.

Unlike the other three return codes, the redirect packet verdict requires an additional parameter that specifies the redirection target, which is set through a helper function before the program exits. The redirect functionality can be used (1) to transmit the raw packet out a different network interface (including virtual interfaces connected to virtual machines), (2) to pass it to a different CPU for further processing, or (3) to pass it directly to a special userspace socket address family (`AF_XDP`). These different packet paths are shown as solid lines in Figure 1. The decoupling of the return code and the target parameter makes redirection a flexible forwarding mechanism, which can be extended with additional target types without requiring any special support from either the XDP programs themselves, or the device drivers implementing the XDP hooks. In addition, because the redirect parameter is implemented as a map lookup (where the XDP program provides the lookup key), redirect targets can be changed dynamically without modifying the program.

3.2 The eBPF Virtual Machine

XDP programs run in the Extended BPF (eBPF) virtual machine. eBPF is an evolution of the original BSD packet filter (BPF) [2] which has seen extensive use in various packet filtering applications over the last decades. BPF uses a register-based virtual machine to describe filtering actions. The original BPF virtual machine has two 32-bit registers and understands 22 different instructions. eBPF extends the number of registers to eleven, and increases register widths to 64 bits. The 64-bit registers map one-to-one to hardware registers on the 64-bit architectures supported by the kernel, enabling efficient just-in-time (JIT) compilation into native machine code. Support for

compiling (restricted) C code into eBPF is included in the LLVM compiler infrastructure [27].

eBPF also adds new instructions to the eBPF instruction set. These include arithmetic and logic instructions for the larger register sizes, as well as a *call* instruction for function calls. eBPF adopts the same calling convention as the C language conventions used on the architectures supported by the kernel. Along with the register mapping mentioned above, this makes it possible to map a BPF call instruction to a single native call instruction, enabling function calls with close to zero additional overhead. This facility is used by eBPF to support helper functions that eBPF programs can call to interact with the kernel while processing, as well as for function calls within the same eBPF program.

While the eBPF instruction set itself can express any general purpose computation, the verifier (described in Section 3.4 below) places limitations on the programs loaded into the kernel to ensure that the user-supplied programs cannot harm the running kernel. With this in place, it is safe to execute the code directly in the kernel address space, which makes eBPF useful for a wide variety of tasks in the Linux kernel, not just for XDP. Because all eBPF programs can share the same set of maps, this makes it possible for programs to react to arbitrary events in other parts of the kernel. For instance, a separate eBPF program could monitor CPU load and instruct an XDP program to drop packets if load increases above a certain threshold.

The eBPF virtual machine supports dynamically loading and re-loading programs, and the kernel manages the life cycle of all programs. This makes it possible to extend or limit the amount of processing performed for a given situation, by adding or completely removing parts of the program that are not needed, and re-loading it atomically as requirements change. The dynamic loading of programs also makes it possible to express processing rules directly in program code, which for some applications can increase performance by replacing lookups into general purpose data structures with simple conditional jumps.

3.3 BPF Maps

eBPF programs are executed in response to an event in the kernel (a packet arrival, in the case of XDP). Each time they are executed they start in the same initial state, and they do not have access to persistent memory storage in their program context. Instead, the kernel exposes helper functions giving programs access to *BPF maps*.

BPF maps are key/value stores that are defined upon loading an eBPF program, and can be referred to from within the eBPF code. Maps exist in both global and per-CPU variants, and can be shared, both between different eBPF programs running at various places in the kernel, as well as between eBPF and userspace. The map types include generic hash maps, arrays and radix trees, as well as specialised types containing pointers to eBPF programs (used for tail calls), or redirect targets, or even pointers to other maps.

Maps serve several purposes: they are a persistent data store between invocations of the same eBPF program; a global coordination tool, where eBPF programs in one part of the kernel can update state that changes the behaviour in another; and a communication mechanism between userspace programs and the kernel eBPF programs, similar to the communication between control plane and data plane in other programmable packet processing systems.

3.4 The eBPF Verifier

Since eBPF code runs directly in the kernel address space, it can directly access, and potentially corrupt, arbitrary kernel memory. To prevent this from happening, the kernel enforces a single entry point for loading all eBPF programs (through the `bpf()` system call). When loading an eBPF program it is first analysed by the in-kernel eBPF verifier. The verifier performs a static analysis of the program byte code to ensure that the program performs no actions that are unsafe (such as accessing arbitrary memory), and that the program will terminate. The latter is ensured by disallowing loops and limiting the maximum program size. The verifier works by first building a directed acyclic graph (DAG) of the control flow of the program. This DAG is then verified as follows:

First, the verifier performs a depth-first search on the DAG to ensure it is in fact acyclic, i.e., that it contains no loops, and also that it contains no unsupported or unreachable instructions. Then, in a second pass, the verifier walks all possible paths of the DAG. The purpose of this second pass is to ensure that the program performs only safe memory accesses, and that any helper functions are called with the right argument types. This is ensured by rejecting programs that perform load or call instructions with invalid arguments. Argument validity is determined by tracking the state of all registers and stack variables through the execution of the program.

The purpose of this register state tracking mechanism is to ensure that the program performs no out of bounds memory accesses *without knowing in advance what the valid bounds are*. The bounds cannot be known because programs must process data packets which vary in size; and similarly, the contents of maps are not known in advance, so it is not known whether a given lookup will succeed. To deal with this, the verifier checks that the program being loaded does its own bounds checking before dereferencing pointers to packet data, and that map lookups are checked for `NULL` values before being dereferenced. This approach leaves the program writer in control of how checks are integrated into the processing logic, and what to do in the error path.

To track data access, the verifier tracks data types, pointer offsets and possible value ranges of all registers. At the beginning of the program, `R1` contains a pointer to the *context* metadata object, `R10` is a stack pointer, and all other registers are marked as not initialised. At each execution step, register states are updated based on the operations performed by the program. When a new value is stored in a register, that register inherits the state variables from

the source of the value. Arithmetic operations will affect the possible value ranges of scalar types, and the offsets of pointer types. The widest possible range is stored in the state variables, e.g., if a one-byte load is performed into a register, that register's possible value range is set to `0-255`. Branches in the instruction graph will update the register state according to the logical operation contained in the branch. For example, given a comparison such as `"R1 > 10"`, the verifier will set the maximum value of `R1` to `10` in one branch, and the minimum value to 11 in the other.

Using this range information stored in the state variables, it is possible for the verifier to predict the ranges of memory that each load instruction can potentially access, and ensure that only safe memory accesses are performed. For packet data access this is done by looking for comparisons with the special `data_end` pointer that is available in the context object; for values retrieved from a BPF map the data size in the map definition is used; and for values stored on the stack, accesses are checked against the data ranges that have previously been written to. Furthermore, restrictions are placed on pointer arithmetic, and pointers cannot generally be converted to integer values. Any eBPF program that performs operations that the verifier cannot prove are safe, are simply rejected at load time. In addition to this, the verifier also uses the range information to enforce aligned memory accesses.

It should be noted that the purpose of the verifier is to protect the internals of the kernel from being exposed to malicious or buggy eBPF programs, not to ensure that the programs perform their designated function in the most efficient way possible. That is, an XDP program can slow down the machine by performing excessive processing (up to the maximum program size), and it can corrupt network packets if written incorrectly. Loading programs requires administrative (root) privileges for this reason, and it is up to the eBPF programmer to prevent these types of bugs, and to the system administrator to decide which programs to load on the system.

3.5 Example XDP program

To showcase the features described above, Listing 1 shows an example of a simple XDP program. The program will parse packet headers, and reflect all UDP packets by swapping the source and destination MAC addresses and sending the packet back out the interface it came in on. While this is obviously a very simple example, the program does feature most of the components of an XDP program that is useful in the real world. Specifically:

- A BPF map is defined (lines 1–7) for keeping statistics of the number of processed packets. The map is keyed on IP protocol number and each value is simply a packet count (updated in lines 55–57). A userspace program can poll this map to output statistics while the XDP program is running.
- Pointers to the start and end of the packet data is read from the context object (lines 27–28), to be used for direct packet data access.

```c
/* map used to count packets; key is IP protocol, value is pkt count */
struct bpf_map_def SEC("maps") rxcnt = {
   .type = BPF_MAP_TYPE_PERCPU_ARRAY,
   .key_size = sizeof(u32),
   .value_size = sizeof(long),
   .max_entries = 256,
};
/* swaps MAC addresses using direct packet data access */
static void swap_src_dst_mac(void *data)
{
   unsigned short *p = data; unsigned short dst[3];
   dst[0] = p[0];   dst[1] = p[1];   dst[2] = p[2];
   p[0]   = p[3];   p[1]   = p[4];   p[2]   = p[5];
   p[3]   = dst[0]; p[4]   = dst[1]; p[5]   = dst[2];
}
static int parse_ipv4(void *data, u64 nh_off, void *data_end)
{
   struct iphdr *iph = data + nh_off;
   if (iph + 1 > data_end)
      return 0;
   return iph->protocol;
}

SEC("xdp1") /* marks main eBPF program entry point */
int xdp_prog1(struct xdp_md *ctx)
{
   void *data_end = (void *)(long)ctx->data_end;
   void *data = (void *)(long)ctx->data;
   struct ethhdr *eth = data; int rc = XDP_DROP;
   long *value; u16 h_proto; u64 nh_off; u32 ipproto;

   nh_off = sizeof(*eth);
   if (data + nh_off > data_end)
      return rc;
   h_proto = eth->h_proto;

   /* check VLAN tag; could be repeated to support double-tagged VLAN */
   if (h_proto == htons(ETH_P_8021Q) || h_proto == htons(ETH_P_8021AD)) {
      struct vlan_hdr *vhdr;
      vhdr = data + nh_off;
      nh_off += sizeof(struct vlan_hdr);
      if (data + nh_off > data_end)
         return rc;
      h_proto = vhdr->h_vlan_encapsulated_proto;
   }

   if (h_proto == htons(ETH_P_IP))
      ipproto = parse_ipv4(data, nh_off, data_end);
   else if (h_proto == htons(ETH_P_IPV6))
      ipproto = parse_ipv6(data, nh_off, data_end);
   else
      ipproto = 0;

   /* lookup map element for ip protocol, used for packet counter */
   value = bpf_map_lookup_elem(&rxcnt, &ipproto);
   if (value)
      *value += 1;

   /* swap MAC addrs for UDP packets, transmit out this interface */
   if (ipproto == IPPROTO_UDP) {
      swap_src_dst_mac(data);
      rc = XDP_TX;
   }
   return rc;
}
```

Listing 1: Example XDP program. The program parses packet headers, swaps source and destination MAC addresses for all UDP packets, and sends them back out the same interface. A packet counter is kept per IP protocol number. Adapted from xdp2_kern.c, which is distributed with the kernel source code.

- Checking against the data_end pointer ensures that no data is read out of bounds (lines 19, 33 and 42). The verifier ensures correctness even across pointer copies (as in lines 19–20).
- The program must handle any packet parsing itself, including things such as VLAN headers (lines 37–45).
- Direct packet data access is used to modify the packet headers (lines 12–14).
- The map lookup helper function exposed by the kernel (called on line 55). This is the only real function call in the program; all other functions are inlined on compilation, including helpers like `htons()`.
- The final packet verdict is communicated by the program return code (line 64).

When the program is installed on an interface, it is first compiled into eBPF byte code, then checked by the verifier. The notable things checked by the verifier in this case are (a) the absence of loops, and the total size of the program, (b) that all direct packet data accesses are preceded by appropriate bounds checking (c) that the sizes of parameters passed to the map lookup function matches the map definition, and (d) that the return value from the map lookup is checked against NULL before it is accessed.

3.6 Summary

The XDP system consists of four major components: (1) The XDP device driver hook which is run directly after a packet is received from the hardware. (2) The eBPF virtual machine which is responsible for the actual program execution (and is also used for executing programs in other parts of the kernel). (3) BPF maps, which allow programs running in various parts of the kernel to communicate with each other and with userspace. And (4) The eBPF verifier, which ensures programs do not perform any operations that can harm the running kernel.

These four components combine to create a powerful environment for writing custom packet processing applications, that can accelerate packet processing in essential paths, while integrating with the kernel and making full use of its existing facilities. The performance achievable by these applications is the subject of the next section.

4 Performance evaluation

In this section we present our performance evaluation of XDP. As mentioned in Section 2, there are quite a few existing systems for high-performance packet processing, and benchmarking all of them is not feasible in the scope of this paper. Instead, we note that DPDK is the existing solution that achieves the highest performance [17], and compare against that as a baseline for the

current state of the art in high-speed software packet processing (using the `testpmd` example application shipped with the 18.05 release of DPDK). We focus on the raw packet processing performance, using synthetic benchmarks, and also compare against the performance of the Linux kernel network stack, to show the performance improvements offered by XDP in the same system. In the next section, we supplement these raw performance benchmarks with some examples of real-world applications implemented on top of XDP, to demonstrate their feasibility within the programming model.

For all benchmarks, we use a machine equipped with a hexa-core Intel Xeon E5-1650 v4 CPU running at 3.60GHz, which supports Intel's Data Direct I/O (DDIO) technology allowing the networking hardware Direct Memory Access (DMA) system to place packet data directly in the CPU cache. The test machine is equipped with two Mellanox ConnectX-5 Ex VPI dual-port 100Gbps network adapters, which are supported by the *mlx5* driver. We use the TRex packet generator [28] to produce the test traffic. The test machine runs a pre-release of version 4.18 of the Linux kernel. To help others reproduce our results, we make available the full details of our setup, along with links to source code and the raw test data, in an online repository [29].

In our evaluation, we focus on three metrics:

- Packet drop performance. To show the maximum packet processing performance, we measure the performance of the simplest possible operation of dropping the incoming packet. This effectively measures the overhead of the system as a whole, and serves as an upper bound on the expected performance of a real packet processing application.
- CPU usage. As mentioned in the introduction, one of the benefits of XDP is that it scales the CPU usage with the packet load, instead of dedicating CPU cores exclusively to packet processing. We quantify this by measuring how CPU usage scales with the offered network load.
- Packet forwarding performance. A packet processing system that cannot forward packets has limited utility. Since forwarding introduces an additional complexity compared to the simple processing case (e.g., interacting with more than one network adapter, rewriting link-layer headers, etc.), a separate evaluation of forwarding performance is useful. We include both throughput and latency in the forwarding evaluation.

We have verified that with full-sized (1500 bytes) packets, our system can process packets at line-speed (100 Gbps) on a single core that is half-idle. This makes it clear that the challenge is processing many *packets* per second, as others have also noted [19]. For this reason, we perform all tests using minimum-sized (64 bytes) packets, and measure the maximum number of packets per second the system can process. To measure how performance scales with the number of CPU cores, we repeat the tests with an increasing number of cores dedicated to packet processing.[23] For XDP and the Linux

[23] The Hyperthreading feature of the CPU is disabled for our experiments, so whenever we refer to the number of active CPU cores, this means the number of physical cores.

network stack (which do not offer an explicit way to dedicate cores to packet processing) we achieve this by configuring the hardware Receive Side Scaling (RSS) feature to steer traffic to the desired number of cores for each test.

As we will see in the results below, our tests push the hardware to its very limits. As such, tuning the performance of the system as a whole is important to realise optimal performance. This includes the physical hardware configuration, configuration of the network adapter features such as Ethernet flow control and receive queue size, and configuration parameters of the Linux kernel, where we for instance disable full preemption and the "retpoline" mitigation for the recent Meltdown and Spectre vulnerabilities. The full details of these configuration issues are omitted here due to space constraints, but are available in the online repository.

The following subsections present the evaluation results for each of the metrics outlined above, followed by a general discussion of the performance of XDP compared to the other systems. As all our results are highly repeatable, we show results from a single test run (with no error bars) to make the graphs more readable.

4.1 Packet Drop Performance

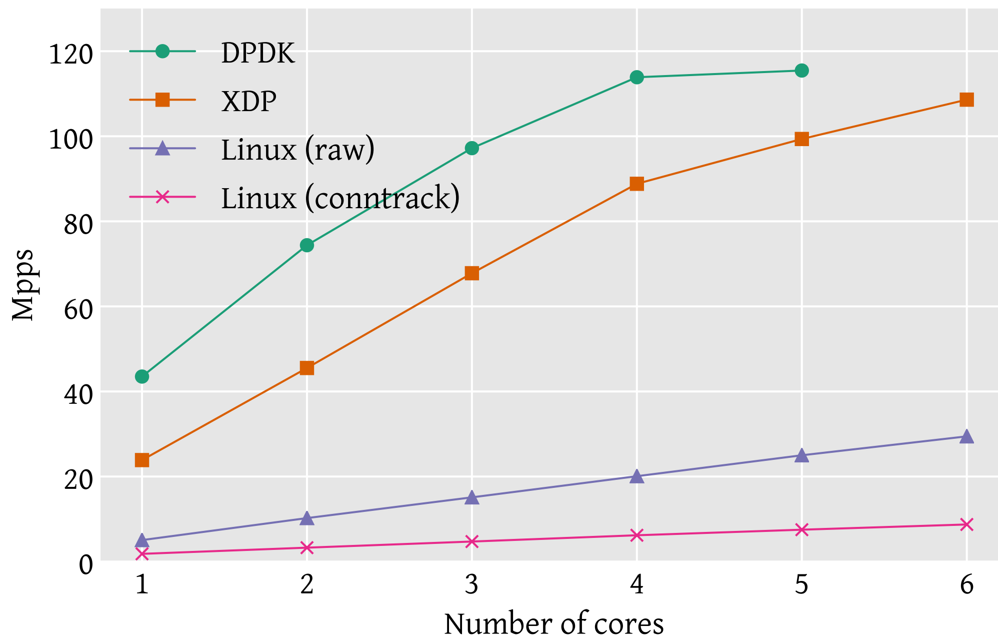


Figure 3: Packet drop performance. DPDK uses one core for control tasks, so only 5 are available for packet processing.

Figure 3 shows the packet drop performance as a function of the number of cores. The baseline performance of XDP for a single core is 24 Mpps, while for DPDK it is 43.5 Mpps. Both scale their performance linearly until they approach the global performance limit of the PCI bus, which is reached at

115 Mpps after enabling PCI descriptor compression support in the hardware (trading CPU cycles for PCI bus bandwidth).

The figure also shows the performance of the Linux networking stack in two configurations: one where we use the “raw” table of the *iptables* firewall module to drop packets, which ensures the earliest possible drop in the network stack processing; and another where we use the connection tracking (*conntrack*) module, which carries a high overhead, but is enabled by default on many Linux distributions. These two modes illustrate the performance span of the Linux networking stack, from 1.8 Mpps of single-core performance with conntrack, up to 4.8 Mpps in raw mode. It also shows that in the absence of hardware bottlenecks, Linux performance scales linearly with the number of cores. And finally, it shows that with its 24 Mpps on a single core, XDP offers a five-fold improvement over the fastest processing mode of the regular networking stack.

As part of this Linux raw mode test, we also measured the overhead of XDP by installing an XDP program that does no operation other than updating packets counters and passing the packet on to the stack. We measured a drop in performance to 4.5 Mpps on a single core, corresponding to 13.3 ns of processing overhead. This is not shown on the figure, as the difference is too small to be legible.

4.2 CPU Usage

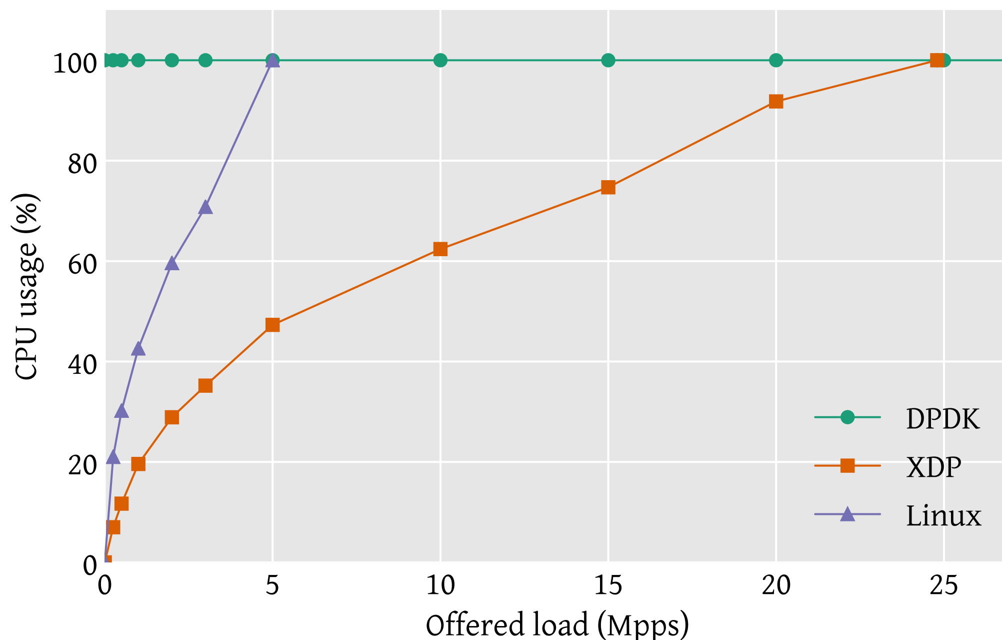


Figure 4: CPU usage in the drop scenario. Each line stops at the method’s maximum processing capacity. The DPDK line continues at 100% up to the maximum performance shown in Figure 3.

We measure the CPU usage of the different tested systems when running the packet drop application on a single CPU core, by recording the percentage of CPU busy time using the `mpstat` system utility. The results of this is shown in Figure 4. The test varies the offered packet load up to the maximum that each system can handle on a single core.

Since DPDK by design dedicates a full core to packet processing, and uses busy polling to process the packets, its CPU usage is always pegged at 100%, which is the green line at the top of the figure. In contrast, both XDP and Linux smoothly scale CPU usage with the offered load, with a slightly larger relative increase in CPU usage at a small offered load level.

The non-linearity of the graph in the bottom-left corner is due to the fixed overhead of interrupt processing. At lower packet rates, the number of packets processed during each interrupt is smaller, leading to higher CPU usage per packet.

4.3 Packet Forwarding Performance

Figure 5 shows packet forwarding performance. The forwarding applications perform a simple Ethernet address rewrite, where the source and destination address of the incoming packet are swapped before the packet is forwarded. This is the minimum rewriting that is needed for packet forwarding to function, so the results represent an upper bound on forwarding performance. Since XDP can forward packets out the same NIC as well as out a different NIC (using two different program return codes), we include both modes in the graph. The DPDK example program only supports forwarding packets through a different interface, so we only include this operating mode in the test. Finally, the Linux networking stack does not support this minimal forwarding mode, but requires a full bridging or routing lookup to forward packets; this lookup is expensive, and since the other applications do not perform it, the results are not directly comparable. For this reason, we omit the Linux networking stack from these results, and instead include the Linux routing performance in our routing use case presented in Section 5.1.

As Figure 5 shows, we again see linear scaling with the number of cores up to a global performance bottleneck. The absolute performance is somewhat lower than for the packet drop case, which shows the overhead of packet forwarding. We also see that the XDP performance improves significantly when packets are sent out on the same interface that they were received on, surpassing the DPDK forwarding performance at two cores and above. The performance difference is primarily due to differences in memory handling: packet buffers are allocated by the device driver and associated with the receiving interface. And so, when the packet is forwarded out a different interface, the memory buffer needs to be returned to the interface that it is associated with.

Looking at forwarding latency, as seen in Table 1, the relative performance of XDP and DPDK for different-NIC forwarding are reflected for the high

Table 1: Packet forwarding latency. Measurement machine connected to two ports on the same NIC, measuring end-to-end latency for 50 seconds with high and low packet rates (100 pps and 1 Mpps).

	Average		Maximum		< 10μs	
	100 pps	1 Mpps	100 pps	1 Mpps	100 pps	1 Mpps
XDP	82μs	7μs	272μs	202μs	0%	98.1%
DPDK	2μs	3μs	161μs	189μs	99.5%	99.0%

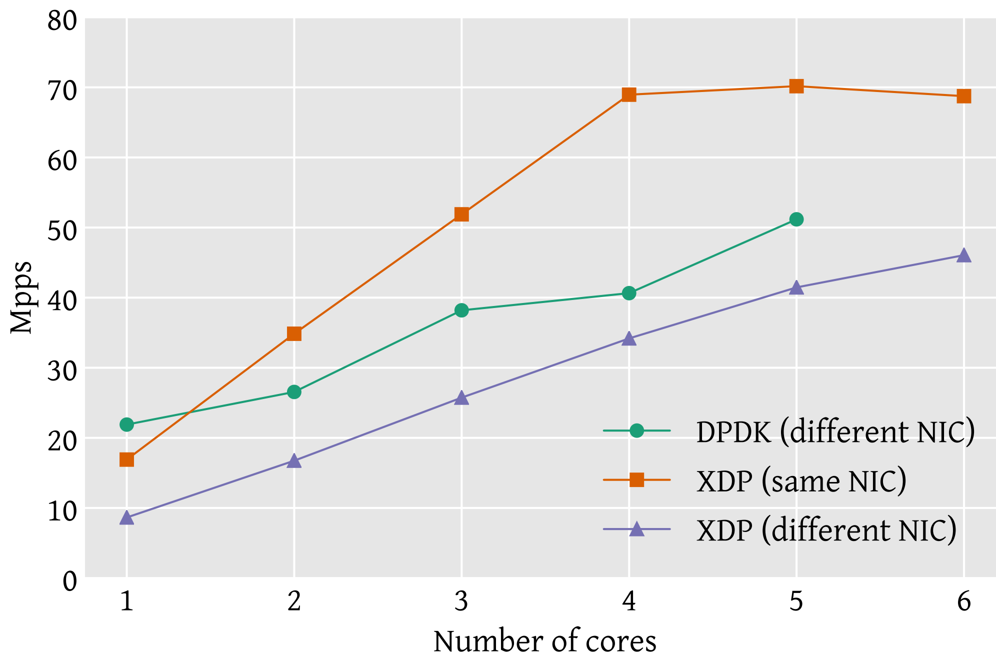


Figure 5: Packet forwarding throughput. Sending and receiving on the same interface takes up more bandwidth on the same PCI port, which means we hit the PCI bus limit at 70 Mpps.

packet rate test (with DPDK showing slightly lower variance as well). However, for low packet rates, the latency of XDP is dominated by the interrupt processing time, which leads to much higher end-to-end latency than DPDK achieves with constant polling.

4.4 Discussion

As we have seen in the previous subsections, XDP achieves significantly higher performance than the regular Linux networking stack. Even so, for most use cases XDP does not quite match the performance of DPDK. We believe this is primarily because DPDK has incorporated more performance optimisations at the lowest level of the code. To illustrate this, consider the packet drop example: XDP achieves 24 Mpps on a single core, which corresponds to 41.6 nanoseconds per packet, while DPDK achieves 43.5 Mpps, or 22.9 nanoseconds

per packet. The difference of 18.7 nanoseconds corresponds to 67 clock cycles on the 3.6 GHz processor in our test machine. Thus, it is clear that every micro-optimisation counts; for example, we measure an overhead of 1.3 nanoseconds for a single function call on our test system. The mlx5 driver performs 10 function calls when processing a single packet, corresponding to 13 of the 18.7 nanoseconds of performance difference between XDP and DPDK.

Some of this overhead is inevitable on a general-purpose operating system such as Linux, as device drivers and subsystems are structured in a way that makes it possible to support a wide variety of systems and configurations. However, we believe that some optimisations are viable. For instance, we have performed an experiment that removed DMA-related function calls that were not needed on our specific hardware from the driver, removing four of the 10 per-packet function calls. This improved the packet drop performance to 29 Mpps. Extrapolating this, removing all function calls would increase performance to 37.6 Mpps. While this is not possible in practice, it is possible to remove some of them, and combining this with other performance optimisations, we believe it is reasonable to expect the performance gap between DPDK and XDP to lessen over time. We see similar effects with other drivers, such as the *i40e* driver for 40 Gbps Intel cards, which achieves full performance up to the NIC hardware performance limit with both XDP and DPDK.[24]

Given the above points, we believe it is feasible for XDP to further decrease the performance delta to DPDK. However, given the benefits of XDP in terms of flexibility and integration with the rest of the system, XDP is already a compelling choice for many use cases; we show some examples of this in the next section.

5 Real-world use cases

To show how the various aspects of XDP can be used to implement useful real-world applications, this section describes three example use cases. These use cases have all seen deployment in one form or another, although we use simplified versions in our evaluation to be able to make the code available. We also refer the reader to [30] for an independent look at some of the challenges of implementing real-world network services in XDP.

The purpose of this section is to demonstrate the feasibility of implementing each use case in XDP, so we do not perform exhaustive performance evaluations against state of the art implementations. Instead, we use the regular Linux kernel stack as a simple performance baseline and benchmark the XDP applications against that. The three use cases are a software router, an inline Denial of Service (DoS) mitigation application and a layer-4 load balancer.

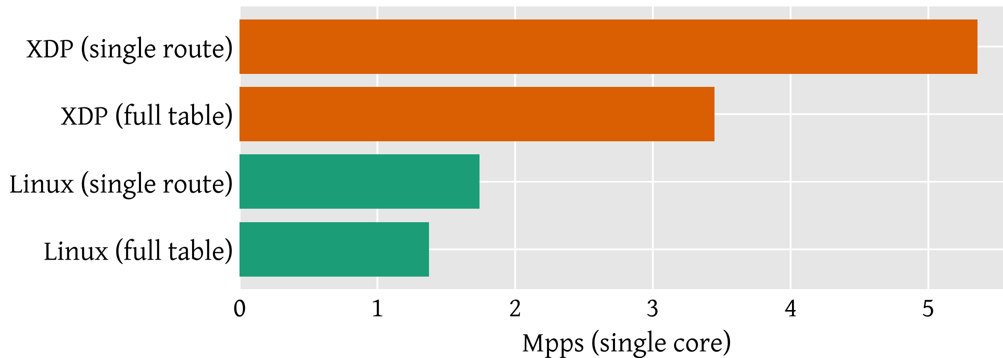


Figure 6: Software routing performance. Since the performance scales linearly with the number of cores, only the results for a single core are shown.

5.1 Software Routing

The Linux kernel contains a full-featured routing table, which includes support for policy routing, source-specific routing, multi-path load balancing, and more. For the control plane, routing daemons such as Bird [31] or FRR [32] implement a variety of routing control plane protocols. Because of this rich ecosystem supporting routing on Linux, re-implementing the routing stack in another packet processing framework carries a high cost, and improving performance of the kernel data plane is desirable.

XDP is a natural fit for this task, especially as it includes a helper function which performs full routing table lookups directly from XDP. The result of the lookup is an egress interface and a next-hop MAC address, which makes it possible for the XDP program to immediately forward the packet if the lookup succeeds. If no next-hop MAC is known (because neighbour lookup has not been performed yet), the XDP program can pass the packet to the networking stack, which will resolve the neighbour, allowing subsequent packets to be forwarded by XDP.

To show the performance of this use case, we use the XDP routing example that is included in the Linux kernel source [33] and compare its performance to routing in the regular Linux network stack. We perform two tests: one with a single route installed in the routing table, and another where we use a full dump of the global BGP routing table from `routeviews.org`. In both cases, all next-hop addresses are set to the address of the test system connected to our egress interface. The full table contains 752,138 distinct routes, and for our tests we generate 4000 random destination IP addresses to make sure we exercise the full table.[25].

The performance of this use case is seen in Figure 6. Using XDP for the forwarding plane improves performance with a factor of 2.5 for a full table

[24]While DPDK uses the drivers in the operating system to assume control of the hardware, it contains its own drivers that are used for the actual packet processing.

[25]Using fewer than 4000 destination IPs, the part of the routing table that is actually used is small enough to be kept in the CPU cache, which gives misleading (better) results. Increasing the number of IPs above 4000 had no additional effects on forwarding performance.

lookup, and a factor of 3 for the smaller routing table example. This makes it feasible to run a software router with a full BGP table at line rate on a 10 Gbps link using a single core (using a conservative estimate of an average packet size of 300 bytes).

5.2 Inline DoS Mitigation

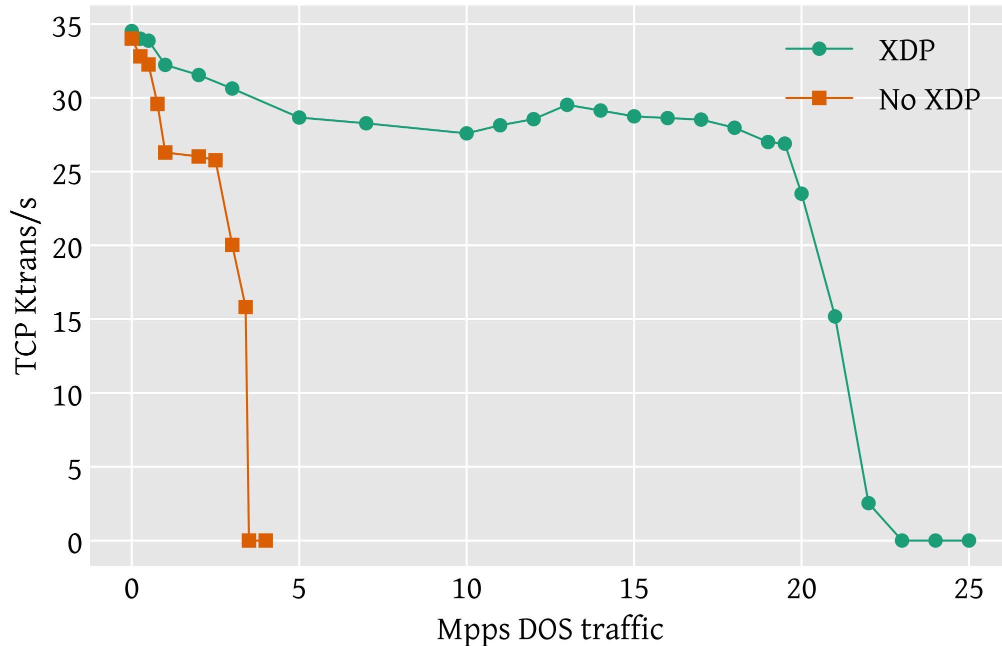


Figure 7: DDoS performance. Number of TCP transactions per second as the level of attack traffic directed at the server increases.

DoS attacks continue to plague the internet, typically in the form of distributed attacks (DDoS attacks) from compromised devices. With XDP, it is possible to deploy packet filtering to mitigate such attacks directly at the application servers, without needing to change applications. In the case of a virtual machine deployment, the filter can even be installed in the hypervisor, and thus protect all virtual machines running on the host.

To show how this could work, we perform a test modelled on the DDoS mitigation architecture used by Cloudflare, which uses XDP as the filtering mechanism [34]. Their Gatebot architecture works by sampling traffic at servers located in distributed Points of Presence (PoPs), collecting it centrally for analysis, and formulating mitigation rules based on the analysis. The mitigation rules take the form of a series of simple checks on the packet payload, which are compiled directly into eBPF code and distributed to the edge servers in the PoPs. Here the code is executed as an XDP program that will drop packets matching the rules, while also updating match counters stored in BPF maps.

To test the performance of such a solution, we use an XDP program that parses the packet headers and performs a small number of tests[26] to identify attack traffic and drop it, and uses the CPU redirect feature to pass all other packets to a different CPU core for processing. To simulate a baseline application load we use the Netperf benchmarking tool [35]. Netperf contains a TCP-based round-trip benchmark, which opens a TCP connection and sends a small payload that is echoed back from the server, repeating as soon as a reply is received. The output is the number of transactions per second, which represents performance of an interactive use case, such as small remote procedure calls.

We run our experiment on a single core, to illustrate the situation where legitimate traffic has to compete for the same hardware resources as attack traffic. We apply a baseline load of 35.000 TCP transactions per second, then simulate the DoS attack by offering an increasing load of small UDP packets matching our packet filter. We measure the TCP transactions performance as the attack traffic volume increases, reporting the average of four test repetitions per data point.

The results of this is shown in Figure 7. Without the XDP filter, performance drops rapidly, being halved at 3 Mpps and effectively zero at just below 3.5 Mpps of attack traffic. However, with the XDP filter in place, the TCP transaction performance is stable at around 28.500 transactions per second until 19.5 Mpps of attack traffic, after which it again drops rapidly. This shows that effective DDoS filtering is feasible to perform in XDP, which comfortably handles 10 Gbps of minimum-packet DoS traffic on a single CPU core. Deploying DDoS mitigation this way leads to increased flexibility, since no special hardware or application changes are needed.

5.3 Load Balancing

For the load balancer use case, we use the XDP component of the Katran load balancer [36] released as open source by Facebook. This works by announcing an IP address for the service, which is routed to the load balancer. The load balancer hashes the source packet header to select a destination application server. The packet is then encapsulated and sent to the application server, which is responsible for decapsulating it, processing the request, and replying directly to the originator. The XDP program performs the hashing and encapsulation, and returns the packet out the same interface on which it was received. It keeps configuration data in BPF maps and implements the encapsulation entirely in the eBPF program.

To test this use case, we configure the Katran XDP program with a fixed number of destination hosts[27], and run it on our test machine. We compare it with the IPVS load balancer that is part of the Linux kernel, which can be configured in the same way. The performance of both is shown in Table 2,

[26] Our example program performs four packet data reads per packet, to parse the outer packet headers and drop packets with a pre-defined UDP destination port number.

[27] We use one virtual IP per CPU core, and 100 destinations per virtual IP.

CPU Cores	1	2	3	4	5	6
XDP (Katran)	5.2	10.1	14.6	19.5	23.4	29.3
Linux (IPVS)	1.2	2.4	3.7	4.8	6.0	7.3

Table 2: Load balancer performance (Mpps).

which shows linear scaling with the number of CPUs, and that XDP offers a performance gain of 4.3x over IPVS.

6 Future directions of XDP

As we have shown above, XDP offers high performance and can be used to implement a variety of real-world use cases. However, this does not mean that XDP is a finished system. On the contrary, as part of the Linux kernel, XDP undergoes continuous improvement. Some of this development effort goes into softening the rough edges that are the inevitable result of XDP being incrementally incorporated into a general purpose operating system kernel. Other efforts continue to push the boundaries of XDP's capabilities. In this section we discuss some of these efforts.

6.1 Limitations on eBPF programs

As mentioned previously, the programs loaded into the eBPF virtual machine are analysed by the eBPF verifier, which places certain limitations on the programs to ensure they do not harm the running kernel. These limitations fall in two categories: (a) Ensuring the program will terminate, which is implemented by disallowing loops and limiting the maximum size of the program. And (b) ensuring the safety of memory accesses, which is done by the register state tracking explained in Section 3.4.

Since the primary function of the verifier is to ensure the safety of the kernel, a conservative approach is taken, and the verifier will reject any program that it cannot prove is safe. This can lead to false negatives, where safe programs are needlessly rejected; reducing such cases is an ongoing effort. The error messages reported by the verifier have also been made friendlier, to make it easier for developers to change their code to fix verification errors when they do occur. Support for function calls in eBPF has recently been added, support for bounded loops is planned, and efficiency improvements of the verifier itself are being worked on, which will allow it to operate on larger programs.

Another limitation of eBPF programs compared to user-space C programs is the lack of a standard library, including things like memory allocation, threading, locking, etc. This is partly alleviated by the life cycle and execution context management of the kernel (i.e., an XDP program is automatically run for each arriving packet), and partly by the helper functions exposed by the kernel.

Finally, only one XDP program can be attached to each networking interface. This can be worked around by cooperation between programs, where the tail call functionality can be used to either dynamically dispatch to different programs depending on packet content, or to chain several programs together.

6.2 User Experience and Debugging

Since an XDP program runs in the kernel, the debugging tools available to a regular userspace program are not generally applicable. Instead, the debugging and introspection features included in the kernel can be applied to XDP (and other eBPF programs). These tools include tracepoints and kprobes [37] as well as the performance counters that are part of the *perf* subsystem [38]. However, developers who are not familiar with the kernel ecosystem may find this ecosystem of kernel-specific tools a limitation. To ease the transition, a variety of tools exist, including the BPF Compiler Collection [39], the *bpftool* introspection program [40] and the *libbpf* library of utility functions [41]. These have already seen significant improvements, but more work is needed in this area.

6.3 Driver Support

Each device driver needs to add support for running XDP programs, by supporting an API exposed by the core networking stack, and support is continuously being added to more and more drivers.[28] However, since features are usually implemented in smaller increments, some care is still needed when selecting hardware to use with XDP, to ensure full support for the features needed for a particular use case. However, since XDP is integrated into the kernel device driver model, it imposes no particular capability constraints on the hardware, which means that full support in all drivers is possible.

As the XDP system has evolved, the need to keep the changes required in drivers to support XDP to a minimum has become increasingly clear, and some steps have been taken in this direction. For instance, support for new targets can be added to the redirection action without any changes needed from the drivers. Finally, the Generic XDP feature [43] allows running XDP programs (at reduced performance) even if the networking driver lacks the proper support, by moving execution into the core networking stack.

6.4 Performance Improvements

As we discussed in Section 4.4, there is still a performance gap between XDP and DPDK in some use cases. Efforts to improve this are ongoing, which includes micro-optimisations of driver code as well as changes to the core XDP code to remove unnecessary operations, and amortise processing costs through improved batching.

[28] At the time of writing Linux 4.18 has XDP support in 12 different drivers, including most high-speed network adapters. For an updated list, see [42].

6.5 QoS and Rate Transitions

Currently, XDP does not implement any mechanism for supporting different Quality of Service (QoS) levels. Specifically, an XDP program receives no back-pressure when attempting to forward packets to a destination that has exhausted its capacity, such as when joining networks with different speeds or other mismatched network characteristics.

While QoS is lacking from XDP, the Linux kernel networking stack features best-in-class Active Queue Management (AQM) and packet scheduling algorithms [44]. Not all of these features are a good fit for the XDP architecture, but we believe that selectively integrating features from the networking stack into XDP is an opportunity to provide excellent support for QoS and AQM in XDP, in a way that can be completely transparent to the packet processing applications themselves. We are planning to explore this further.

6.6 Accelerating Transport Protocols

With XDP we have shown how high-speed packet processing can be integrated cooperatively into the operating system to accelerate processing while making use of existing features of the operating system where it makes sense. XDP focuses on stateless packet processing, but extending the same model to stateful transport protocols such as TCP would provide many of the same performance benefits to applications that require reliable (and thus stateful) transports. Indeed, others have shown that accelerated transport protocols can significantly improve performance relative to the regular operating system stack [45–48].

One of these previous solutions [45] shows that there is significant potential in improving the raw packet processing performance while keeping the in-kernel TCP stack itself. XDP is a natural fit for this, and there has been some initial discussion of how this could be achieved [49]; while far from trivial, this presents an exciting opportunity for expanding the scope of the XDP system.

6.7 Zero-copy to userspace

As mentioned in Section 3.1, an XDP program can redirect data packets to a special socket opened by a user space application. This can be used to improve performance of network-heavy applications running on the local machine. However, in its initial implementation, this mechanism still involves copying the packet data, which negatively affects performance. There is ongoing work to enable true zero-copy data transfer to user space applications through `AF_XDP` sockets. This places some constrains on the memory handling of the network device, and so requires explicit driver support. The first such support was merged into the kernel in the 4.19 release cycle, and work is ongoing to add it to more drivers. The initial performance numbers look promising, showing transfers of upwards of 20 Mpps to userspace on a single core.

6.8 XDP as a building block

Just as DPDK is used as a low-level building block for higher level packet processing frameworks (e.g., [11]), XDP has the potential to serve as a runtime environment for higher-level applications. In fact, we have already started to see examples of applications and frameworks leveraging XDP. Prominent examples include the Cilium security middle-ware [50], the Suricata network monitor [51], Open vSwitch [52] and the P4-to-XDP compiler project [25]. There is even an effort to add XDP as a low-level driver for DPDK [53].

7 Conclusion

We have presented XDP, a system for safely integrating fast programmable packet processing into the operating system kernel. Our evaluation has shown that XDP achieves raw packet processing performance of up to 24 Mpps on a single CPU core.

While this is not quite on par with state of the art kernel bypass-based solutions, we argue that XDP offers other compelling features that more than make up for the performance delta. These features include retaining kernel security and management compatibility; selectively utilising existing kernel stack features as needed; providing a stable programming interface; and complete transparency to applications. In addition, XDP can be dynamically re-programmed without service interruption, and requires neither specialised hardware nor dedicating resources exclusively to packet processing.

We believe that these features make XDP a compelling alternative to all-or-nothing kernel bypass solutions. This belief is supported by XDP's adoption in a variety of real-world applications, some of which we have shown examples of. Furthermore, the XDP system is still evolving, and we have outlined a number of interesting developments which will continue to improve it in the future.

Acknowledgements

XDP has been developed by the Linux networking community for a number of years, and the authors would like to thank everyone who has been involved. In particular, Alexei Starovoitov has been instrumental in the development of the eBPF VM and verifier; Jakub Kicinski has been a driving force behind XDP hardware offloading and the bpftool utility; and Björn Töpel and Magnus Karlsson have been leading the AF_XDP and userspace zero-copy efforts.

We also wish to extend our thanks to the anonymous reviewers, and to our shepherd Srinivas Narayana, for their helpful comments.

References

[1] "Data plane development kit," Linux Foundation, 2018. https://www.dpdk.org/

[2] S. McCanne and V. Jacobson, "The BSD Packet Filter: A New Architecture for User-level Packet Capture," in *USENIX winter*, vol. 93, 1993.

[3] L. Rizzo and G. Lettieri, "Vale, a switched ethernet for virtual machines," in *Proceedings of the 8th international conference on Emerging networking experiments and technologies*. ACM, 2012.

[4] S. Han *et al.*, "PacketShader: a GPU-accelerated software router," in *ACM SIGCOMM Computer Communication Review*, vol. 40, no. 4. ACM, 2010.

[5] D. Kirchner *et al.*, "Augustus: a CCN router for programmable networks," in *Proceedings of the 3rd ACM Conference on Information-Centric Networking*. ACM, 2016.

[6] P. M. Santiago del Rio *et al.*, "Wire-speed statistical classification of network traffic on commodity hardware," in *Proceedings of the 2012 Internet Measurement Conference*. ACM, 2012.

[7] R. B. Mansilha *et al.*, "Hierarchical content stores in high-speed ICN routers: Emulation and prototype implementation," in *Proceedings of the 2nd ACM Conference on Information-Centric Networking*. ACM, 2015.

[8] P. Emmerich *et al.*, "Moongen: A scriptable high-speed packet generator," in *Proceedings of the 2015 Internet Measurement Conference*. ACM, 2015.

[9] S. Han *et al.*, "MegaPipe: A new programming interface for scalable network I/O," in *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI '12)*, 2012.

[10] T. Marian, K. S. Lee, and H. Weatherspoon, "NetSlices: scalable multi-core packet processing in user-space," in *Proceedings of the eighth ACM/IEEE symposium on Architectures for networking and communications systems*. ACM, 2012.

[11] L. Linguaglossa *et al.*, "High-speed software data plane via vectorized packet processing," Telecom ParisTech, Tech. Rep., 2017.

[12] R. Morris *et al.*, "The Click modular router," *ACM SIGOPS Operating Systems Review*, vol. 33, no. 5, 1999.

[13] M. Dobrescu *et al.*, "RouteBricks: exploiting parallelism to scale software routers," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 2009.

[14] B. Pfaff *et al.*, "The design and implementation of Open vSwitch," in *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI '15)*, 2015.

[15] "PF_RING ZC (Zero Copy)," Ntop project, 2018. https://www.ntop.org/products/packet-capture/pf_ring/pf_ring-zc-zero-copy/

[16] "OpenOnload," Solarflare Communications Inc, 2018. https://www.openonload.org/

[17] S. Gallenmüller *et al.*, "Comparison of frameworks for high-performance packet IO," in *Proceedings of the Eleventh ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ser. ANCS '15. IEEE Computer Society, 2015, pp. 29–38.

[18] J. Networks, "Juniper Contrail Virtual Router," 2018. https://github.com/Juniper/contrail-vrouter

[19] L. Rizzo, "Netmap: a novel framework for fast packet I/O," in *21st USENIX Security Symposium (USENIX Security 12)*, 2012.

[20] L. Deri, "Modern packet capture and analysis: Multi-core, multi-gigabit, and beyond," in *the 11th IFIP/IEEE International Symposium on Integrated Network Management (IM)*, 2009.

[21] S. Peter *et al.*, "Arrakis: The operating system is the control plane," *ACM Transactions on Computer Systems (TOCS)*, vol. 33, no. 4, 2016.

[22] J. Martins *et al.*, "ClickOS and the art of network function virtualization," in *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*. USENIX Association, 2014.

[23] J. W. Lockwood *et al.*, "NetFPGA–an open platform for gigabit-rate network switching and routing," in *IEEE International Conference on Microelectronic Systems Education*. IEEE, 2007.

[24] P. Bosshart *et al.*, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, 2014.

[25] "p4c-xdp," VMWare, 2018. https://github.com/vmware/p4c-xdp

[26] J. Kicinski and N. Viljoen, "eBPF/XDP hardware offload to SmartNICs," in *NetDev 1.2 - The Technical Conference on Linux Networking*, 2016.

[27] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. IEEE Computer Society, 2004.

[28] "TRex traffic generator," Cisco, 2018. https://trex-tgn.cisco.com/

[29] T. Høiland-Jørgensen *et al.*, “Xdp-paper online appendix,” 2018. https://github.com/tohojo/xdp-paper

[30] S. Miano *et al.*, “Creating complex network service with eBPF: Experience and lessons learned,” in *IEEE International Conference on High Performance Switching and Routing*, 2018.

[31] “BIRD internet routing daemon,” CZ.nic, 2018. https://bird.network.cz/

[32] “FRRouting,” The Linux Foundation, 2018. https://frrouting.org/

[33] D. Ahern, “XDP forwarding example,” 2018. https://elixir.bootlin.com/linux/v4.18-rc1/source/samples/bpf/xdp_fwd_kern.c

[34] G. Bertin, “Xdp in practice: integrating xdp in our ddos mitigation pipeline,” in *NetDev 2.1 - The Technical Conference on Linux Networking*, 2017.

[35] R. Jones, “Netperf,” Open source benchmarking software, 2018. http://www.netperf.org/

[36] “Katran source code repository,” Facebook, 2018. https://github.com/facebookincubator/katran

[37] “Linux tracing technologies,” Linux documentation authors, 2018. https://www.kernel.org/doc/html/latest/trace/index.html

[38] “perf: Linux profiling with performance counters,” perf authors, 2018. https://perf.wiki.kernel.org/index.php/Main_Page

[39] “BCC BPF Compiler Collection,” IO Visor, 2018. https://www.iovisor.org/technology/bcc

[40] “bpftool manual,” bpftool authors, 2018. https://elixir.bootlin.com/linux/v4.18-rc1/source/tools/bpf/bpftool/Documentation/bpftool.rst

[41] “libbpf source code,” libbpf authors, 2018. https://elixir.bootlin.com/linux/v4.18-rc1/source/tools/lib/bpf

[42] “BPF and XDP reference guide,” Cilium Authors, 2018. https://cilium.readthedocs.io/en/latest/bpf/

[43] D. S. Miller, “Generic XDP,” 2017. https://git.kernel.org/torvalds/c/b5cdae3291f7

[44] T. Høiland-Jørgensen, P. Hurtig, and A. Brunstrom, “The Good, the Bad and the WiFi: Modern AQMs in a residential setting,” *Computer Networks*, vol. 89, Oct. 2015.

[45] K. Yasukata *et al.*, “StackMap: Low-latency networking with the OS stack and dedicated NICs,” in *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. USENIX Association, 2016, pp. 43–56.

[46] I. Marinos, R. N. Watson, and M. Handley, "Network stack specialization for performance," in *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4. ACM, 2014, pp. 175–186.

[47] A. Belay *et al.*, "IX: A protected dataplane operating system for high throughput and low latency," in *Proceedings of the 11th USENIX Symposium on Operating System Design and Implementation (OSDI '14)*. USENIX, 2014.

[48] E. Jeong *et al.*, "mTCP: a highly scalable user-level TCP stack for multicore systems." in *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI '14)*, vol. 14, 2014, pp. 489–502.

[49] T. Herbert, "Initial thoughts on TXDP," 2016. https://www.spinics.net/lists/netdev/msg407537.html

[50] "Cilium software," Cilium, 2018. https://github.com/cilium/cilium

[51] "Suricata - eBPF and XDP," Suricata, 2018. https://suricata.readthedocs.io/en/latest/capture-hardware/ebpf-xdp.html

[52] W. Tu, "[ovs-dev] AF_XDP support for OVS," 2018. https://mail.openvswitch.org/pipermail/ovs-dev/2018-August/351295.html

[53] Q. Zhang, "[dpdk-dev] PMD driver for AF_XDP," 2018. http://mails.dpdk.org/archives/dev/2018-February/091502.html

Paper IX

Flent

The FLExible Network Tester

Reprinted from

11th EAI International Conference on Performance Evaluation Methodologies and Tools (VALUETOOLS 2017), December 5–7, 2017, Venice, Italy

"I've experiments to run, there is research to be done
On the people who are still alive."

"Still Alive" — End credits of the Portal game

Flent
The FLExible Network Tester

Toke Høiland-Jørgensen, Carlo Augusto Grazia, Per Hurtig and Anna Brunstrom

toke.hoiland-jorgensen@kau.se, carloaugusto.grazia@unimore.it, per.hurtig@kau.se, anna.brunstrom@kau.se

Abstract

Running network performance experiments on real systems is essential for a complete understanding of protocols and systems connected to the internet. However, the process of running experiments can be tedious and error-prone. In particular, ensuring reproducibility across different systems is difficult, and comparing different test runs from an experiment can be non-trivial.

In this paper, we present a tool, called Flent, designed to make experimental evaluations of networks more reliable and easier to perform. Flent works by composing well-known benchmarking tools to, e.g., run tests consisting of several bulk data flows combined with simultaneous latency measurements. Tests are specified in source code, and several common tests are included with the tool. In addition, Flent contains features to automate test runs, collect relevant metadata and interactively plot and explore datasets.

We showcase Flent's capabilities by performing a set of experiments evaluating the new BBR congestion control algorithm, using Flent's capabilities to reproduce experiments both in a controlled testbed and across the public internet. Our evaluation reveals several interesting features of BBR's performance.

1 Introduction

Running properly managed experimental evaluations of networks and network services can be tedious and error-prone, which is part of the reason why new network technologies are often primarily (or even exclusively) evaluated by simulation. In this work we present a tool that is designed to aid researchers in increasing reliability and making it easier to run tests on real network hardware.

There are several reasons why it is valuable to run experiments instead of simulations. The main reason is also the most obvious: simulations are necessarily idealised, and may be inaccurate; real-world systems simply behave *differently* than simulations. Another reason to prefer experiments on real systems is the sheer pace of development of, especially, open source operating systems. Linux, in particular, has seen sweeping changes to its network stack over the last years, in many aspects completely changing its behaviour. These changes mean that the assumptions underlying new research need to be verified against actual systems running on the internet.

We present a tool designed to work towards the goal of making testing more reliable and easier to carry out. This tool, called Flent ("the FLExible Network Tester"), works by composing well-known benchmarking tools to (for example) run tests consisting of several bulk data flows combined with simultaneous latency measurements or measure application traffic while loading the link with background flows. Tests are specified in source code, and several common tests are included with the tool. In addition, Flent contains features to automate test runs, collect metadata and auxiliary data sets, and to interactively plot data collected from experiments.

To showcase Flent's capabilities, we perform an experimental evaluation of the BBR congestion control algorithm [1] for TCP. BBR was released recently, and has only been subject of a few independent evaluations. As such, we consider an evaluation of BBR both timely, and appropriate for showing the capabilities of Flent. We evaluate the performance of BBR in a controlled testbed with different Active Queue Management (AQM) algorithms on the bottleneck link, as well as over a link on the public internet. The evaluation reveals several interesting features of BBR, including its ability to function without using drops as a sign of congestion.

The rest of the paper is structured as follows: Section 2 elaborates on the difficulties that we seek to alleviate, and Section 3 describes Flent and how it addresses them. Section 4 showcases Flent's capabilities by performing an experimental evaluation of the BBR congestion control algorithm. Section 5 summarises related work and, finally, Section 6 contains the concluding summary.

2 Experimental challenges

The key difficulties that the Flent testing tool is designed to address, are reproducibility of experiments, testbed configuration and automation, and storage and analysis of measurement data. The rest of this section outlines each of these in turn.

2.1 Reproducing experiments

Reproducing experiments is important for verifiability, for both the researcher, and for independent reproduction by others. However, actually creating reproducible experiments is challenging [2].

Part of the reason for this is the complexity of coordinating different test tools. Since many network benchmarking tools are single-purpose, running different tools at the same time can be necessary when creating complex test scenarios. Often, ad-hoc scripting is the tool of choice when combining test tools, but that can be error-prone and tedious, and usually results in duplication of effort between different test scenarios and deployments. The more ad-hoc the test configuration and setup is, the harder it is to ensure reproducibility of the tests.

2.2 Testbed configuration and test automation

Experiments often involve testbeds comprising several physical devices set up to emulate the desired network topology and characteristics, and the configuration of these devices must be managed. This includes correctly configuring network interfaces, applying the algorithm(s) under test, etc. Additionally, the configuration must be verifiable after the tests have run, so that it is possible to certify that a data set corresponds to a particular configuration.

This process can be error-prone, especially as the number of configuration parameters that vary between test runs increase. In addition, configuration can fail, either from human error or from (unchecked) failures in the configuration process, so automation is important in both application and subsequent collection.

Finally, being able to automatically run a test series, including applying configuration between runs, significantly reduces the difficulty of experimental work, and makes it practical to test a larger set of variables in a single setting.

2.3 Storing and analysing measurement data

As the number of experiments grows, storing the measurement data and relating it to the right tested configuration becomes harder. This is exacerbated by the previous issue of coordinating several benchmarking tools with possibly different output formats. A standardised way is needed to manage these different benchmarking tool outputs, and extracting the meaningful data points for analysis.

In addition, gathering relevant metadata can be extremely helpful in verifying the test setup and avoiding spurious errors from faulty configurations that make the test data invalid.

3 How Flent helps

Flent is developed specifically to address the difficulties mentioned in the previous section, while also being extensible to address other future use cases. Flent is written in Python and can drive several other well-known network benchmarking tools, most notably Netperf. This section describes how Flent

seeks to address each of the challenges presented in the previous section. A design diagram for Flent is available in the online documentation.[29]

3.1 Reproducing experiments

Flent works by running one or more tests, each defined by a configuration file that specifies which benchmarking tools to run, leveraging well-known tools such as Netperf and Iperf. Several tools can be run simultaneously, or in series, and dependencies can be specified between them (e.g., run one tool once another has finished). Each test defined in Flent has a name and a separate configuration file. This greatly aids reproducibility, as the named tests are available along with the source code, and can be referenced reliably across different systems running Flent. The tests are quite versatile, since larger test suites can be composed of the available named tests. This also allows Flent to work well with other tools that manage tests: Anywhere there's a Python environment and the required underlying benchmarking tools, Flent can run.

When running a test, the output of each testing tool is parsed, and the output data is stored in a common JSON-based format. This makes it easy to create composite tests comprising several different tools, and afterwards directly compare the data collected by the tools. A common example employed in many of the tests included with Flent is running one or more instances of Netperf to produce bulk flows, while simultaneously measuring the end-to-end latency by means of the regular `ping` command.

3.2 Configuration and automation

Flent manages configuration of the test environment by including support for running arbitrary scripts at the start and end of each test in a series. We have found that in the small to medium-sized testbed environments targeted by Flent, scripting is the most flexible choice for configuring nodes. To ensure correctness, these scripts must run before each test invocation, and any failures must be detected. To this end, the batch feature simply provides a facility to run arbitrary commands before and after each test, and (optionally) abort the test if any of the commands fail. Combined with the metadata gathered by Flent at each test-run, this is an effective configuration management and verification facility, without the need to include platform-specific configuration code in Flent itself.

In addition, Flent has built-in batch run capabilities, making it possible to specify a series of test runs to be run in sequence, while supporting inheritance and recursive expansion of variables and config sections to facilitate configuration reuse. By means of this facility, extensive test suites can be built from the named tests.

[29] Flent (and its online documentation) is available at `https://flent.org/`.

3.3 Storing and analysing measurement data

Flent automatically gathers metadata from the host running the test (and optionally from remote hosts via SSH), and stores the metadata in the data file along with the test data. This means that a single data file can capture a complete test run and be easily transferred to another system for further analysis. In addition, auxiliary data sets can be captured along with the main data series, including queueing statistics from the Linux qdisc layer, socket statistics reported by the operating system during the test, CPU usage, WiFi rate statistics and more. As with the metadata, these auxiliary data sets can be captured from both the local machine running the test, and from instrumented remote devices such as intermediate routers.

Flent also contains an extensive analysis facility, which reads already produced test files and produces plots of the data. A graphical user interface makes it easy to flip between plots of different test runs. Tests can define detailed plots (such as the raw timeseries data of throughput during the test run), as well as aggregate plot types (CDFs, box plots, etc), and it is possible to show several test runs side by side, as well as to combine them into aggregate plots. The interactive plotting feature makes for a powerful analysis tool in the exploratory phases of experimental work. Additionally, the tool can also produce final high-quality graphs for publication: The example plots in the next section are all produced by Flent's built-in plotting facilities.

Many plot types are supported, allowing exploration to be performed directly from within Flent. Should this not be sufficient, it is also possible to export the data to other formats: there's a CSV export feature in Flent itself, and the JSON data format is readily parsable by other tools.

4 Showcasing Flent: A look at the BBR congestion control

In this section, we showcase the capabilities of Flent to effectively run experiments, by evaluating the BBR congestion control algorithm [1] for TCP. This algorithm was proposed recently, and has garnered significant interest in the research community. BBR is designed to ensure high utilisation at all bandwidths without inducing unnecessary queueing latency (also known as bufferbloat). It does this by continuously estimating delivery rate and path RTT and adjusting its sending bandwidth accordingly. Every eight roundtrips, BBR will enter a one-RTT probing phase to probe for more bandwidth, followed by a drain phase where sending bandwidth is temporarily lowered to allow buffers to drain. BBR also employs packet pacing, where packets are sent at a constant rate over the whole RTT, instead of being bursted out in window-sized bursts as traditional TCP does.

We believe it is relevant to evaluate how BBR performs in the presence of another technology that has been shown to be effective against bufferbloat: Active Queue Management (AQM) algorithms installed at the bottleneck link. Performing such an evaluation is exactly what Flent excels at, since it involves

applying well-known testing methodology (bufferbloat tests) to a new scenario (BBR).

To also showcase how we can perform the same tests in different environments and compare the results, we repeat out tests over the public internet. In both evaluations, we compare the performance of BBR with the well-known CUBIC congestion control. We focus on the application level goodput and the end-to-end latency under load as our metrics of interest. In the testbed experiments, we apply the two state of the art AQM algorithms CoDel [3] and PIE [4], which both work by dropping packets before the queue fills up, to signal TCP to slow down. We also apply the FQ-CoDel [5] hybrid AQM and packet scheduling algorithm, which combines CoDel with a flow scheduler to also provide flow isolation, fairness and low latency for sparse flows.

4.1 Experimental setup

For our testbed evaluation, we re-use the testbed from a previous study of AQM algorithms [6]. The testbed consists of five nodes, connected as depicted in Figure 1. All nodes are regular x86 PCs with Intel Core 2 CPUs and Intel 82571EB network controllers, running Debian Jessie with a backported Linux kernel version 4.11. We configure the bottleneck to be 10 Mbps and the baseline RTT between client and server to be 50 ms. This emulates a connection to a server on the internet over a residential internet connection.

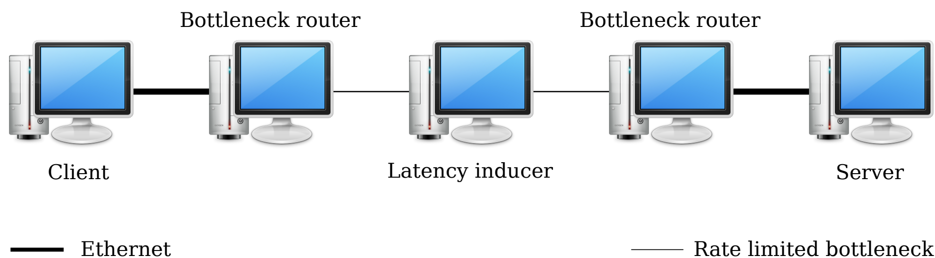


Figure 1: Testbed setup.

Flent runs at the node labelled 'Client' and runs tests against the 'Server' node. We run three different tests included in the Flent source distribution: A TCP download test and a TCP upload test (each running one TCP flow in the respective direction combined with a periodic ICMP Ping to measure latency), and the Realtime Response Under Load (RRUL) test [7], which is designed by the bufferbloat community as a stress test to show the presence of bufferbloat. The latter test runs a total of eight simultaneous TCP flows (four in each direction) while simultaneously measuring latency using both UDP and ICMP packets.

For the test running over the public internet, we run the same suite of tests, but between a test machine located at the University of Modena and Reggio Emilia in Italy and another machine located at Karlstad University in Sweden. The path characteristics between the two endpoints are unknown before the experiment is conducted.

We use the batch facility of Flent to repeat all tests 15 times (for the internet tests) and 30 times (for the testbed tests). We use the interactive plotting interface to explore individual test runs and point out interesting features, and the aggregate plotting facilities to combine all the test runs into meaningful metrics for the tested algorithms. Detailed instructions for replicating the experiments are available online.[30]

4.2 Testbed results

In the testbed results, we first examine the behaviour of a single flow with a FIFO queue, using the TCP upload test.[31] We use Flent's default timeseries graph of throughput and ping latency over the duration of the test to get an idea of how the two different congestion control algorithms work. These graphs are shown in Figure 2. We clearly see how the latency increases to around a full second when using CUBIC; and when the queue overflows we see spikes in both latency and goodput as delivery of date to the application first slows down, then catches up after packets have been retransmitted.

In contrast to this, BBR manages a level throughput and low latency for most of the time, punctuated by spikes in latency every 10 seconds when the algorithm probes for more bandwidth. This shows BBR working as intended and preventing most of the bufferbloat on the bottleneck link from affecting the application. This comes at only a small cost in bandwidth: BBR achieves an average of 9.35 Mbps, while CUBIC achieves 9.55 Mbps.

Turning to the AQM algorithms, one feature of BBR's behaviour is that since it does not interpret loss as a sign of congestion, it is not being controlled only by the AQM. This can be seen in the data in two ways: In the drop behaviour of the AQMs, and in the relative performance of the different AQM algorithms. The number of packets dropped by the CoDel AQM is seen in Figure 3. This is captured from the bottleneck router by the auxiliary data collection facility in Flent. From the figure, it is very clear that CoDel works as intended for the CUBIC flow: After a small burst of drops during TCP slow start (seen at top-left of the graph), CoDel tunes itself to a drop rate that keeps CUBIC oscillating around the bottleneck bandwidth: around one drop per two seconds in this case.

In contrast to this, for the BBR flow, CoDel keeps increasing the drop rate in an attempt to get the queue under control, whenever BBR increases the rate in the probe phase. This is seen in the crosses on the lower part of Figure 3 that look like they are aligned vertically above one another. Actually, these are not completely vertical, but rather show a sharply increasing drop rate over a very short time (this is barely visible in the figure, but can be clearly seen when zooming in on the plot). Between the probe episodes, periods of quiescence appear, where the AQM does not drop any packets.

[30] See `https://www.cs.kau.se/tohojo/flent/`.

[31] Since the bottleneck is symmetrical, the TCP download test shows identical behaviour in this scenario.

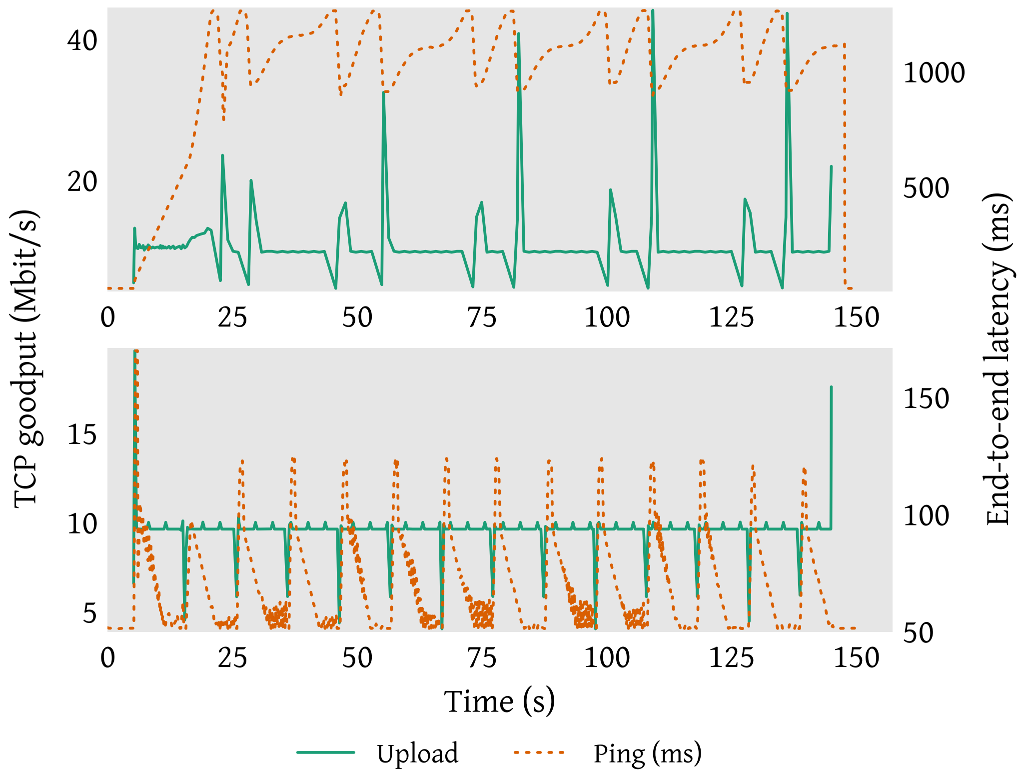


Figure 2: Single flow and ping, FIFO queue. CUBIC (top) and BBR (bottom).

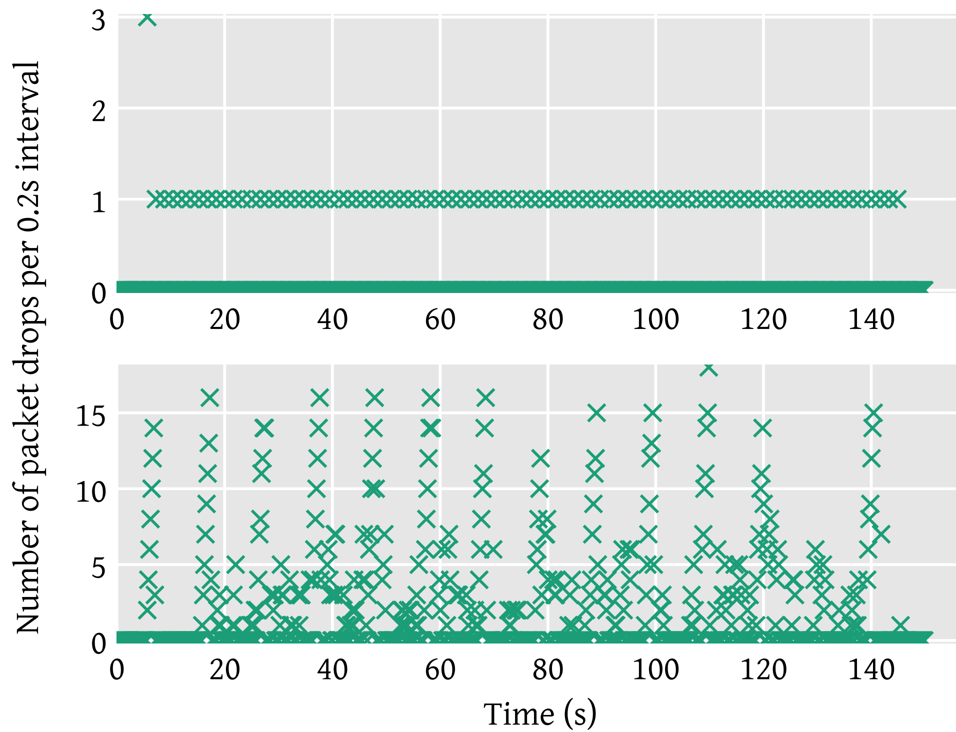


Figure 3: Number of packets dropped by CoDel per measurement interval (0.2 seconds). CUBIC (top) and BBR (bottom).

When looking at the latency distribution, it is clear that BBR is less influenced by the AQM than CUBIC is. To show this, we turn to one of the aggregate plots available in Flent. Figure 4 shows a CDF of the latency measurements across all test repetitions for the CoDel and PIE AQMs for both CUBIC and BBR. Consistent with earlier evaluations of the AQMs [6], CoDel achieves lower delay than PIE, since it tends to drop more aggressively. However, because this difference is more pronounced when using CUBIC (the leftmost and rightmost lines), it means that the relative performance of BBR and CUBIC is reversed depending on the AQM: With CoDel, the queueing delay is lower when using CUBIC that when using BBR, while the reverse is true for PIE. Another interesting feature is the shape of the tail of latency measurements: For CUBIC, the AQMs have a few but quite high spikes of latency at the 99th percentile. For BBR, these do not appear, but a thicker tail, starting at the 80th percentile, of latency measurements up to 125 ms is clearly visible.

The FQ-CoDel hybrid AQM/fairness queueing algorithm shows almost no induced latency for either congestion control, consistent with earlier results. This is not shown in Figure 4 as it would obscure the lines of the other algorithms; however, the FQ-CoDel results are included for the RRUL test considered next.

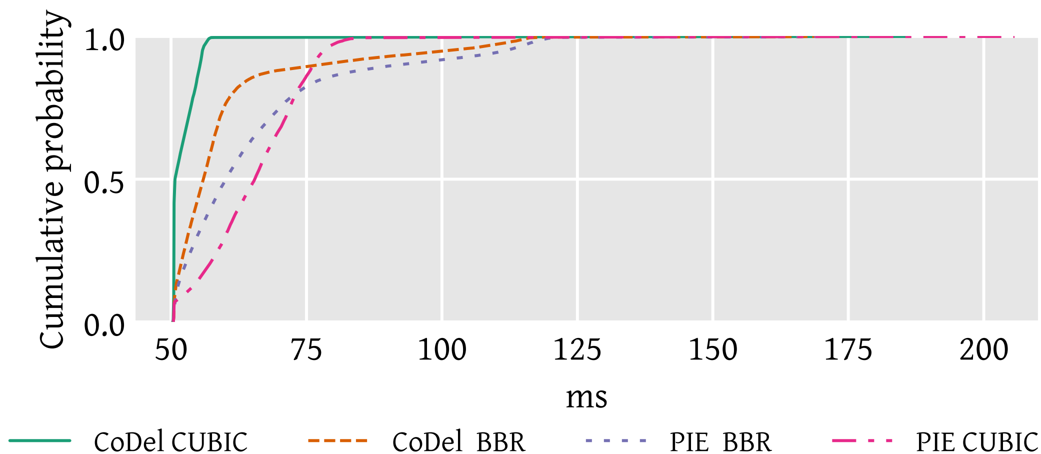


Figure 4: Latency distributions with a single flow for the CoDel and PIE AQM algorithms.

To evaluate a scenario with higher load on the link, the RRUL test is an excellent tool. An overview of the aggregate behaviour for the different combinations of AQMs and congestion control algorithms using the RRUL test is shown in Figure 5.[32] This plot is a so-called "ellipsis plot", which is another of the plot types included in Flent.[33] This plot is excellent for summarising a lot of data in a compact representation. The latency axis is flipped to make "better" values be up and to the right, to fit the intuition of

[32]The CUBIC FIFO case is omitted as that shows latency so high that it would squeeze the rest of the figure to the point of making it illegible.

[33]The use of this type of graph has not been invented for Flent; it was pioneered for visualising bandwidth/latency tradeoffs by Winstein in [8].

laymen. The dots for each data series is the median values, while the ellipses are 1-σ ellipses of the values on the two axes (so larger ellipses indicate higher variance and the angle of the ellipsis shows the direction of covariance between the variables).

From the figure, the same trends are visible as before: The behaviour of BBR does not change significantly depending on the AQM. As the RRUL test has a larger number of active flows, the latency induced by BBR is higher; and so CUBIC induces less latency than BBR with both PIE and CoDel, but BBR achieves slightly higher throughput. Finally, the superior performance of the FQ-CoDel AQM is clearly visible.

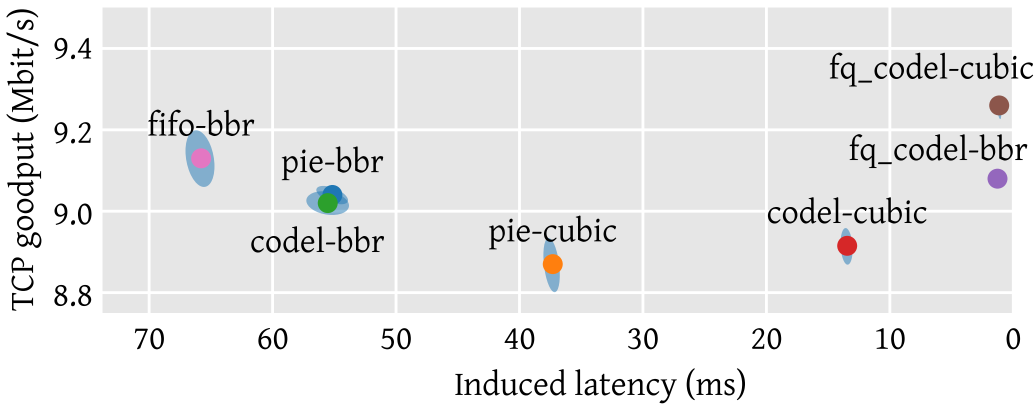


Figure 5: Ellipsis plot of throughput and latency for the RRUL test.

In summary, our testbed evaluation clearly shows several interesting features of BBR: It achieves a significantly lower latency than CUBIC on a FIFO queue, although the intermittent probing behaviour gives significant latency spikes. In addition, the induced latency is higher when many flows are active, and AQM algorithms do not impact the performance as much as they do for CUBIC. However, the AQMs will sharply increase their drop rates in response to BBRs probing behaviour. While this does not impact the behaviour of BBR itself in this test, it has the potential to impact other flows sharing the bottleneck negatively.

4.3 Public internet results

To test how BBR performs outside a controlled environment, we set out to repeat out experiments "in the wild". Because the tests we performed are built-in to Flent, repeating the tests was simply a matter of installing the tool, transferring the configuration file for batch run to another machine, and making a few adjustments (such as the target host for the tests). We performed these repeated tests over the public internet between our two universities. The path characteristics were not known in advance, but the test data allows us to infer several things of interest, as we will see below.

The initial one-flow test is shown in Figure 6. BBR shows consistent performance around 110Mbps of throughput, while CUBIC has an initial

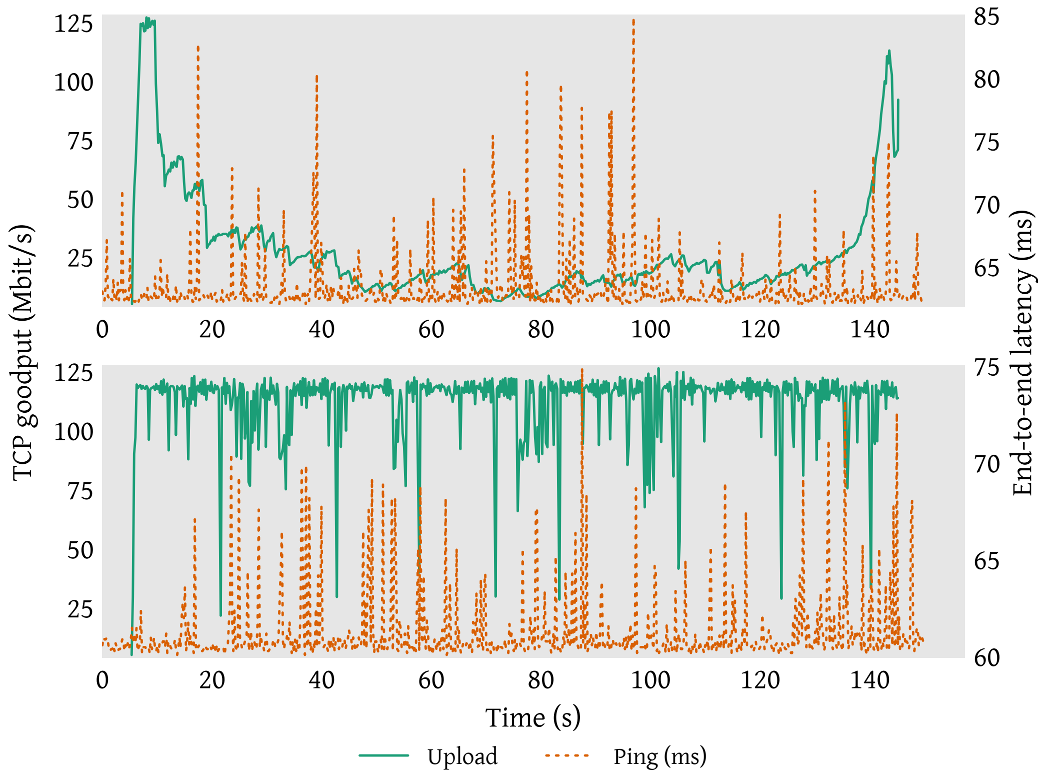


Figure 6: Initial test run over the internet. CUBIC (top) and BBR (bottom).

spike at around the same speed, but quickly becomes erratic and achieves low overall throughput (20 Mbps mean throughput for CUBIC over the whole test, 110 Mbps for BBR).

Since the CUBIC throughput decreases over a series of events, we hypothesise that the bottleneck router is shallowly buffered, and other users sharing the bottleneck causes our flow to experience a lot of packet drops. To confirm that the drops in throughput correspond to window reduction events, we examine the TCP congestion window data (as reported by the operating system and captured as an auxiliary data series by Flent).

This is shown in Figure 7 for both the flow depicted above, and for another flow from a separate test we conducted at night (where the network load is likely to be lower, which should lead to fewer overflow-induced drops). From this figure, we clearly see the characteristic CUBIC window increase and decrease. We also see significantly better performance for the experiment conducted at night (dashed line); the average throughput for this run is 115 Mbps, on par with what BBR achieves. However, we still see several consecutive loss events, which strengthens our shallow buffer hypothesis.

Next, we turn to the RRUL test. Figure 8 shows the aggregate results of the whole test series for both BBR and CUBIC, and showcases the box plot type also featured in Flent. We also added a third contender: CUBIC with packet pacing (labelled "P. CUBIC" in the graph), to see how much of BBR's performance comes from pacing.

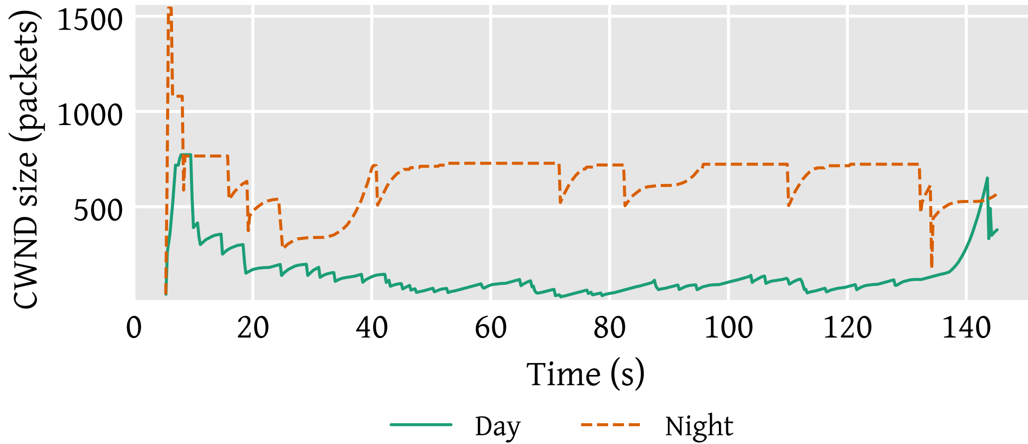


Figure 7: TCP CUBIC congestion window size, daytime and nighttime tests.

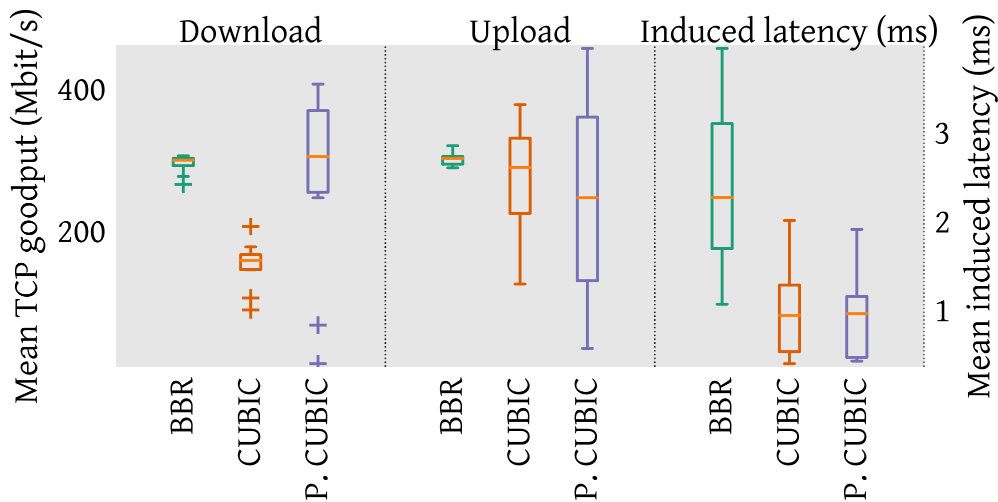


Figure 8: Aggregate values for all RRUL tests over the internet. "P. CUBIC" is CUBIC with packet pacing enabled.

From the figure, we see that having more than one active flow significantly increases the total bandwidth available. We also see that pacing does indeed help CUBIC achieve better throughput in one direction, while it is less conclusive in the other. In either case, BBR consistently performs significantly better, keeping aggregate throughput steady for all tests, at a slight cost in latency.

In summary, we have repeated our tests from the testbed over the public internet, and confirmed several features of BBR. In particular, not treating every drop as a sign of congestion allows BBR to achieve better performance in the face of what appears to be a congested bottleneck link with shallow buffers, although possibly to the detriment of competing flows. We have also shown how to use the data gathered by Flent to infer properties of the bottleneck link, such as the congestion levels at different times of day.

5 Related work

The difficulties of properly constructing and performing experiments, and of reporting accurately on the results of them are not limited to experiments conducted on real hardware. For instance, Kurkowski et al [2] found that many simulation studies in the MANET research community suffered from a series of common errors, many of which are related to those discussed here (e.g., lack of reproducibility and ambiguous initial configuration).

Turning to test tools, the TEACUP system [9] is a test automation framework created specifically to test TCP implementations. It differs from Flent in that its focus is on managing an entire testbed infrastructure. This means that it has more features for configuration, but also makes more assumptions on topology and the nature of the testbed than Flent does. Additionally, while TEACUP offers graphing and analysis of test results, these are more limited, and there is no interactive GUI to explore the data. Netesto [10] is closer in function to Flent, but also focuses on orchestration of several nodes when running a test, and supports fewer types of traffic.

D-ITG [11] is a traffic generation and test platform with an extensive list of supported traffic profiles. It is geared towards running in a managed testbed and emphasis is on remote management with a separate control network to transfer logging data. As such, unlike Flent, D-ITG does not include facilities to interoperate with other tools and does not offer integrated analysis and plotting tools. Flent can use D-ITG as a benchmarking tool in test definitions.

TEMPEST [12] is a simulation framework that makes it possible to run packet scheduling code from operating system kernels in a simulated environment and evaluate various performance metrics with an accessible graphical user interface. As such, it shares the goal of Flent of making it easier to evaluate real networking code, but takes the approach of porting the code to a simulation environment instead of running experiments in a live environment.

Dummynet [13] and netem [14] are emulation modules that can be used in a real network topology to emulate network features not available in the physical hardware. Flent can be used to run experiments in topologies that include emulated dummynet or netem links. Indeed, we use netem for the purpose of adding latency to the bottleneck link in our own testbed.

Finally, several platforms are available for researchers to run their experiments in extensive testbeds, either isolated or across the public internet [15–18]. Flent can be used to drive experiments on these platforms, further increasing the ease of running experiments.

6 Conclusions and Future Work

We have presented Flent, a tool to facilitate experimental evaluations of networks, specifically designed to deal with commonly encountered issues with running experiments. These issues include creating reproducible tests, storing and analysing data, and test automation and configuration management. Flent tackles each of these issues, and is flexible enough to be widely applicable.

We have showcased Flent through an analysis of the BBR congestion control algorithm, both in a controlled testbed with various active queue management algorithms installed, and in an uncontrolled setting over the public internet. This analysis has shown how BBR functions without reacting to drops as a sign of congestion, and revealed several interesting consequences of this behaviour.

Development of Flent is ongoing, and future plans include improving the ability to run tests from the graphical user interface, as well as adding support for more data sources and low-level test tools. This development is guided by feedback from the online community as well as our own needs when running experiments.

References

[1] N. Cardwell *et al.*, "BBR: congestion-based congestion control," *Communications of the ACM*, vol. 60, no. 2, pp. 58–66, 2017.

[2] S. Kurkowski, T. Camp, and M. Colagrosso, "MANET simulation studies: The incredibles," *ACM SIGMOBILE Mobile Computing and Communications Review*, vol. 9, no. 4, pp. 50–61, Oct. 2005.

[3] K. Nichols and V. Jacobson, "Controlling queue delay," *Communications of the ACM*, vol. 55, no. 7, pp. 42–50, Jul. 2012.

[4] R. Pan *et al.*, "PIE: A lightweight control scheme to address the bufferbloat problem," in *2013 IEEE 14th International Conference on High Performance Switching and Routing (HPSR)*, July 2013, pp. 148–155.

[5] T. Høiland-Jørgensen *et al.*, "The Flow Queue CoDel Packet Scheduler and Active Queue Management Algorithm," RFC 8290 (Experimental), RFC Editor, Jan. 2018.

[6] T. Høiland-Jørgensen, P. Hurtig, and A. Brunstrom, "The Good, the Bad and the WiFi: Modern AQMs in a residential setting," *Computer Networks*, vol. 89, pp. 90–106, Oct. 2015.

[7] D. Taht, "RFC: Realtime Response Under Load (rrul) test specification," Nov 2012. https://www.bufferbloat.net/projects/bloat/wiki/RRUL_Spec/

[8] K. Winstein, "Transport architectures for an evolving internet," Ph.D. dissertation, Massachusetts Institute of Technology, Jun. 2014. https://cs.stanford.edu/~keithw/www/Winstein-PhD-Thesis.pdf

[9] S. Zander and G. Armitage, "TEACUP v0.8 – a system for automated TCP testbed experiments," CAIA, Swinburne Univ. of Tech, Tech. Rep. 150210A, February 2015.

[10] L. Brakmo, "Netesto, a network testing toolkit," in *NetDev 2.1 - The Technical Conference on Linux networking*, apr 2017. https://netdevconf.org/2.1/session.html?brakmo

[11] A. Botta, A. Dainotti, and A. Pescapè, "A tool for the generation of realistic network workload for emerging networking scenarios," *Computer Networks*, vol. 56, no. 15, pp. 3531–3547, 2012.

[12] M. Casoni, C. A. Grazia, and P. Valente, "Tempest: a new test environment for performance evaluation of the scheduling of packets," *Simulation Modelling Practice and Theory*, vol. 49, 2014.

[13] M. Carbone and L. Rizzo, "Dummynet revisited," *ACM SIGCOMM Computer Communication Review*, vol. 40, no. 2, pp. 12–20, March 2010.

[14] S. Hemminger, "netem," 2017. https://wiki.linuxfoundation.org/networking/netem

[15] L. Peterson *et al.*, "Experiences building planetlab," in *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association, 2006, pp. 351–366.

[16] E. Eide, L. Stoller, and J. Lepreau, "An experimentation workbench for replayable networking research," in *Proceedings of the Fourth USENIX Symposium on Networked Systems Design and Implementation (NSDI '07)*, 2007.

[17] V. Bajpai *et al.*, "Global Measurements: Practice and Experience (Dagstuhl Seminar 16012)," *Dagstuhl Reports*, vol. 6, no. 1, pp. 15–33, 2016.

[18] V. Bajpai and J. Schönwälder, "A survey on internet performance measurement platforms and related standardization efforts," *IEEE Communications Surveys & Tutorials*, vol. 17, no. 3, pp. 1313–1341, 2015.

BUFFERBLOAT AND BEYOND

The topic of this thesis is the performance of computer networks in general, and the internet in particular. While network performance has generally improved with time, over the last several years we have seen examples of performance barriers limiting network performance. In this work we explore such performance barriers and look for solutions.

Our exploration takes us through three areas where performance barriers are found: The bufferbloat phenomenon of excessive queueing latency, the performance anomaly in WiFi networks and related airtime resource sharing problems, and the problem of implementing high-speed programmable packet processing in an operating system. In each of these areas we present solutions that significantly advance the state of the art.

The work in this thesis spans all three aspects of the field of computing, namely mathematics, engineering and science. We perform mathematical analysis of algorithms, engineer solutions to the problems we explore, and perform scientific studies of the network itself. All our solutions are implemented as open source software, including both contributions to the upstream Linux kernel, as well as the Flent test tool, developed to support the measurements performed in the rest of the thesis.

DOCTORAL THESIS
Karlstad University Studies, 2018:42
ISBN 978-91-7063-878-7 (Print)
ISBN 978-91-7063-973-9 (pdf)