Computer Networks 89 (2015) 90–106

Contents lists available at ScienceDirect

Computer Networks

journal homepage: www.elsevier.com/locate/comnet

The Good, the Bad and the WiFi: Modern AQMs in a residential setting

Toke Høiland-Jørgensen*, Per Hurtig, Anna Brunstrom

Department of Mathematics and Computer Science, Karlstad University, 651 88 Karlstad, Sweden

ARTICLE INFO

Article history:
Received 16 February 2015
Revised 4 July 2015
Accepted 28 July 2015
Available online 31 July 2015

Keywords:
Active queue management
Fairness queueing
Bufferbloat
Latency
Performance measurement
Wireless networks

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

* Corresponding author. Tel.: +46547001611.
E-mail addresses: toke.hoiland-jorgensen@kau.se (T. Høiland-Jørgensen), per.hurtig@kau.se (P. Hurtig), anna.brunstrom@kau.se (A. Brunstrom).

http://dx.doi.org/10.1016/j.comnet.2015.07.014

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.[1]

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, 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 straightforward, 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 offer very good performance in many cases, but note 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

[1] http://www.cs.kau.se/tohojo/good-bad-wifi/.

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.

The tests are run in a controlled environment consisting of five regular desktop computers, as shown in Fig. 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 practice 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

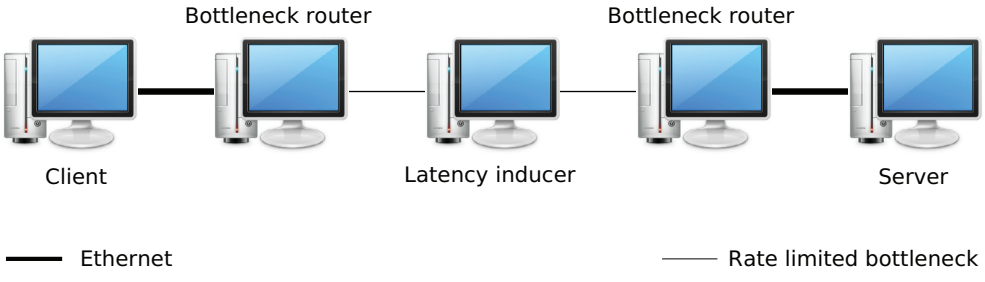


Fig. 1. Physical test setup.

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 have 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.

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.

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.

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

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 s). 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,[2] 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 test the impact of algorithms on performance of real-world application traffic. Each of the steady-state tests is run for 140 s (to minimise the impact of transient behaviour at flow start-up time) and repeated 30 times.

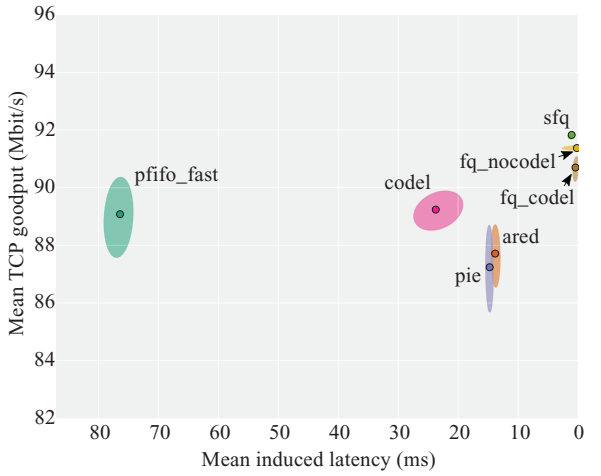
5.1. The real-time response under load test

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.

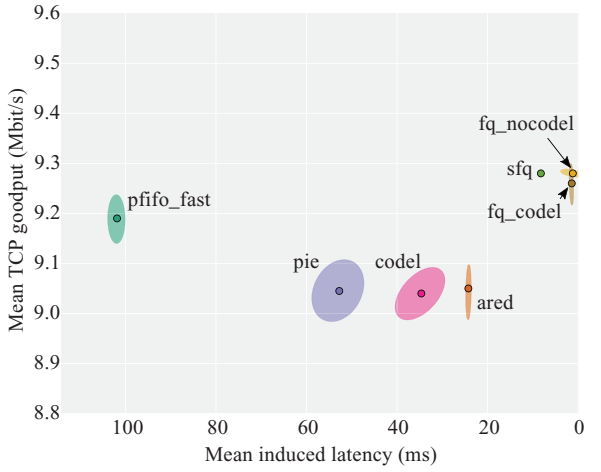
5.1.1. RRUL results

The results for the RRUL test are shown in Fig. 2 as latency-goodput ellipsis graphs. The use of this type of graph was pioneered for visualising bandwidth/latency tradeoffs by

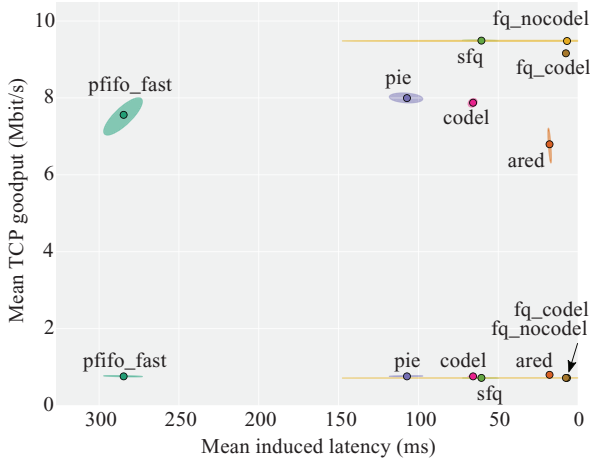
[2] When an overflow condition is detected, fq_codel linearly searches all available queues to find the longest one from which to drop a packet. This has a large impact, mainly by using up a lot of CPU cache. The implementors found this to have acceptable performance as long as it is used as a fallback 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.

Fig. 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.

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 Fig. 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 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 Fig. 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 gives 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.

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 (56 kB 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).[3] We believe these 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.

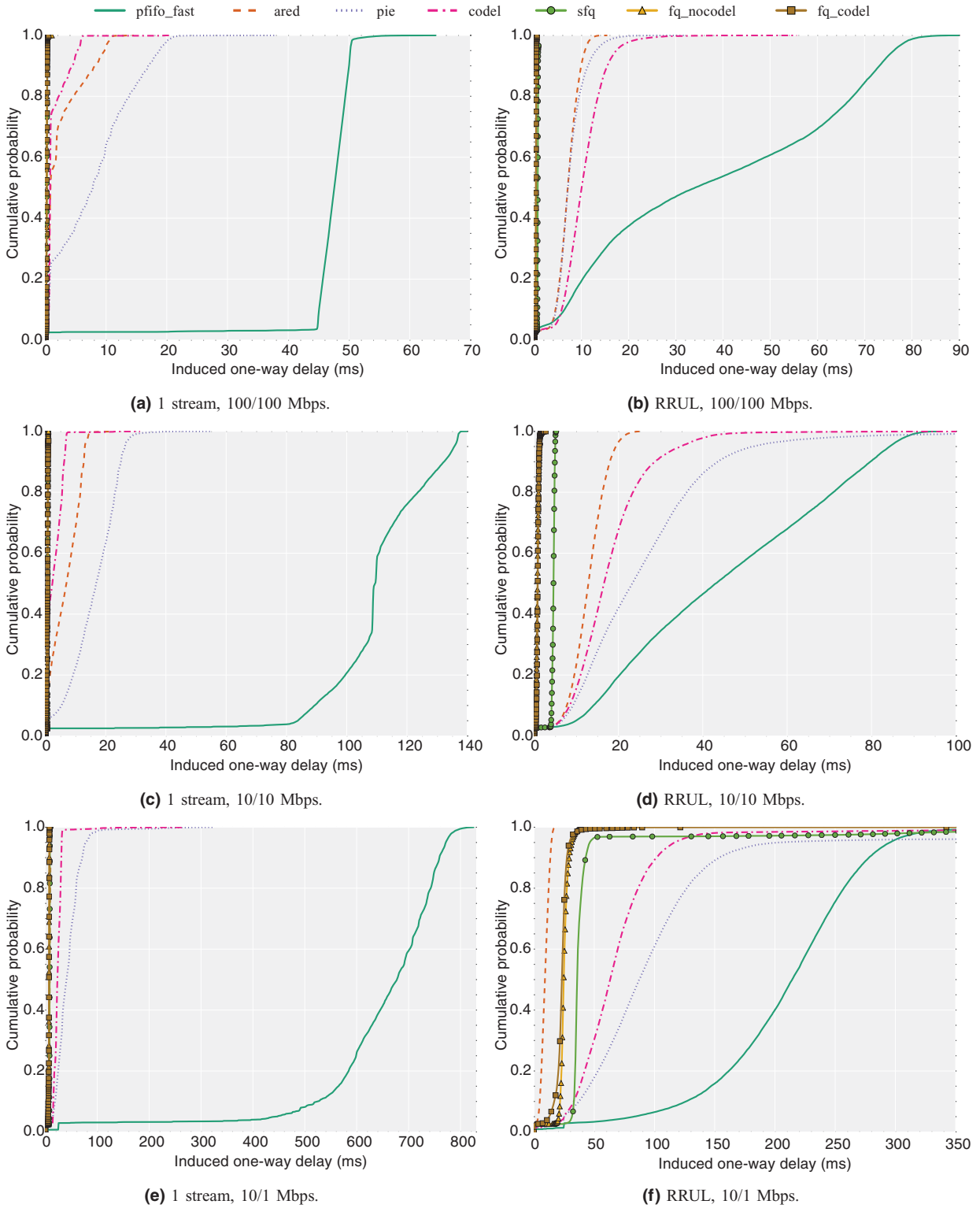
5.3.1. Web results

The results for the web tests are shown in Figs. 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

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



(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.

Fig. 3. VoIP tests 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 Mbps (%)	10 Mbps (%)	100 Mbps (%)
1 stream cross traffic			
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			
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	–	–

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.

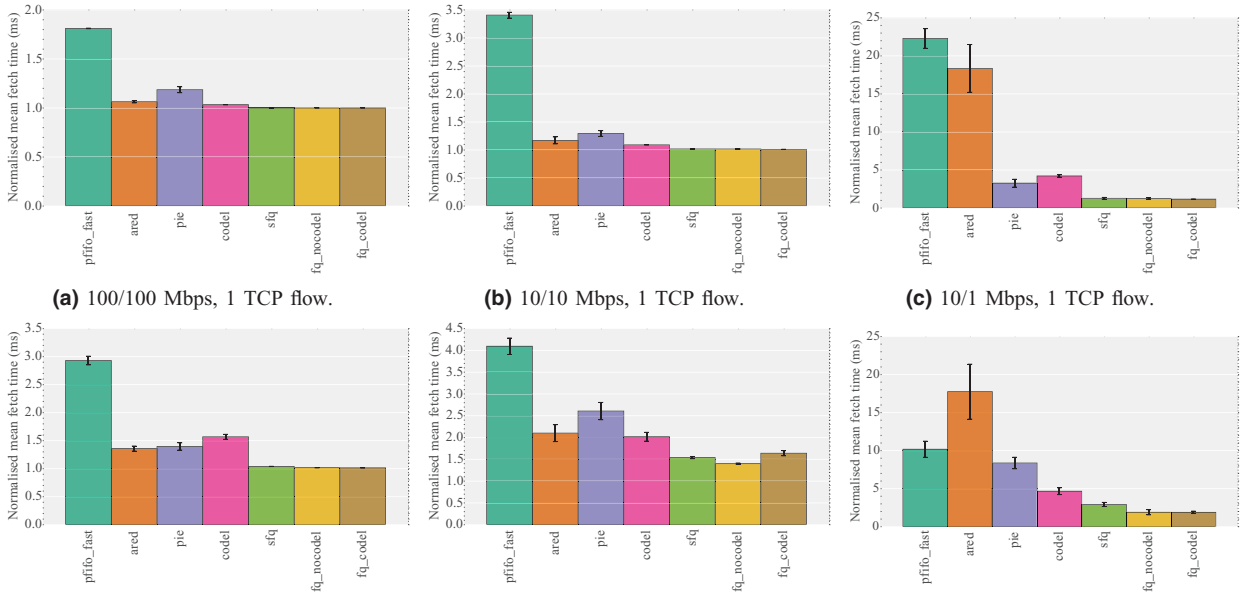
Another interesting feature of the result is that *any* queue management significantly improves this important real-world application performance. The performance differences between the AQM algorithms and the fairness queueing schemes are in many cases less pronounced than in the other tests, since 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.

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.

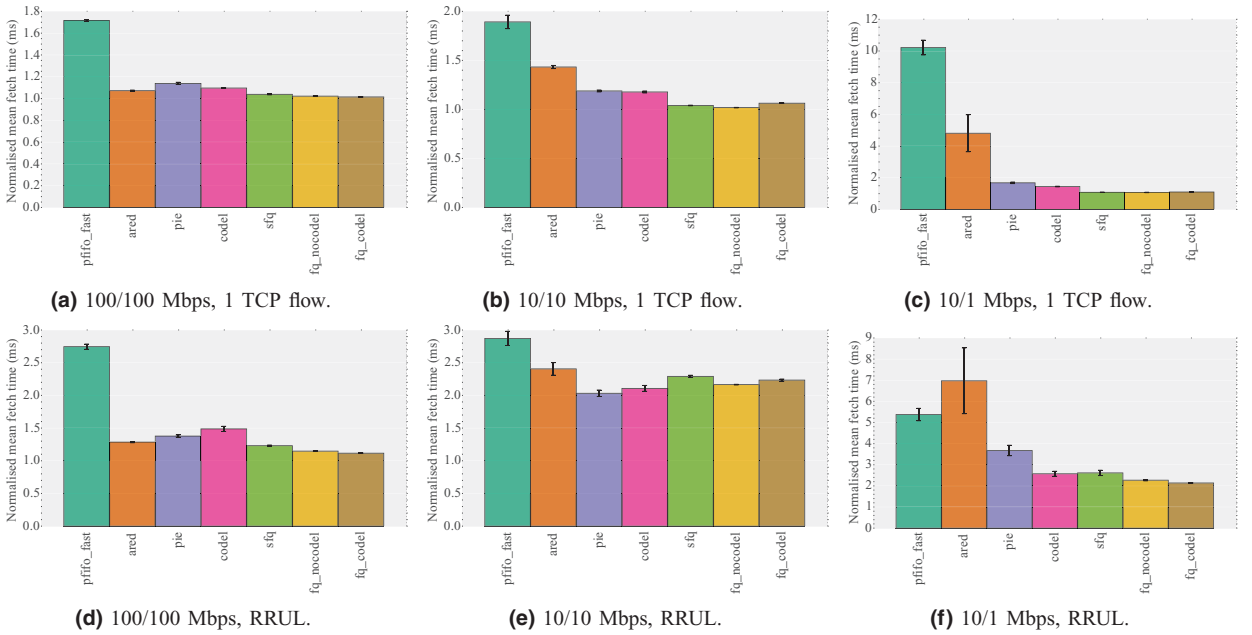
The analysis of the fairness queueing algorithms shows very impressive performance. At no point are the fairness queueing algorithms out-performed by the AQM algorithms, and in most cases fairness queueing outperforms AQM by a large margin. For VoIP traffic in particular, the flow isolation prevents the VoIP flows from experiencing a loss rate that, at the lowest bandwidth, would make any conversation



(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.

Fig. 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.

Fig. 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.

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

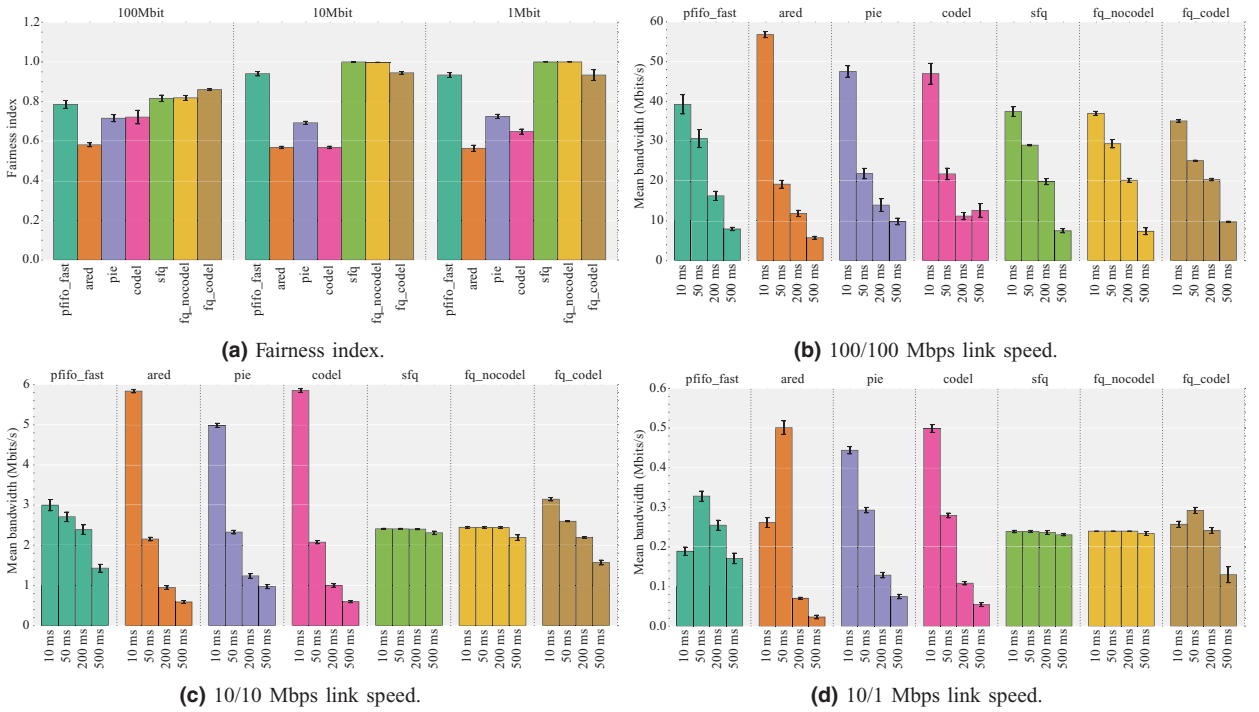
It is well-known that fairness queueing algorithms can improve flow fairness characteristics [35], and indeed it is a design goal for such algorithms (hence the term *fairness queueing*). However, fairness characteristics of pure AQM algorithms are not well understood. In this section, we investigate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

Fig. 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.



(a) Fairness index. **(b)** 100/100 Mbps link speed.

(c) 10/10 Mbps link speed. **(d)** 10/1 Mbps link speed.

Fig. 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.

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, 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.

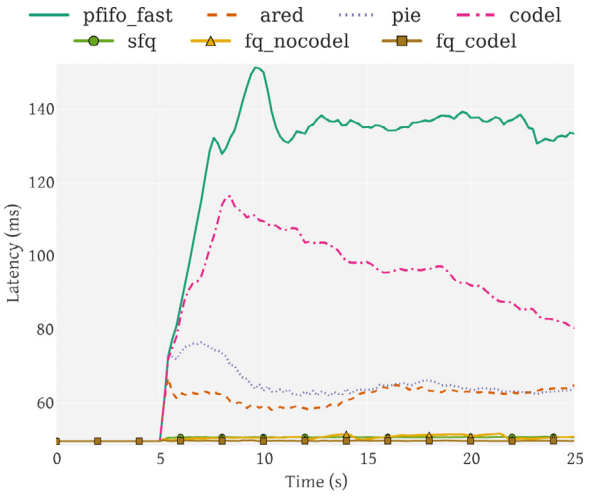
6.2. Transient behaviour

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 nonexistent, 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.

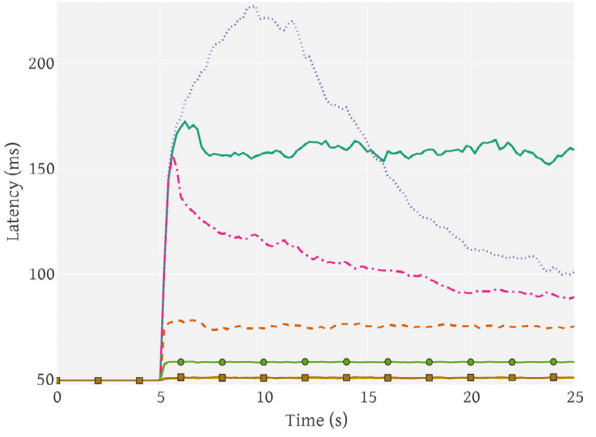
6.2.1. Transient behaviour results

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

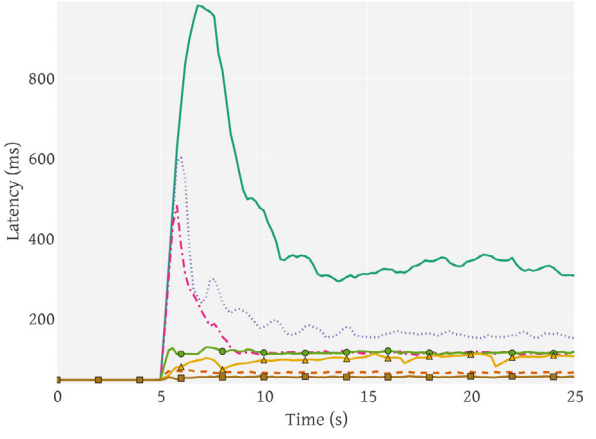
The results show that both CoDel and PIE have severe problems keeping the delay low when the TCP flows start up. At the lower bandwidths, PIE has the worst behaviour, with delay sky-rocketing and even temporarily being higher than for the FIFO queue in the 10 Mbps tests. CoDel fares



(a) 100/100 Mbps link speed.



(b) 10/10 Mbps link speed.



(c) 10/1 Mbps link speed.

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

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

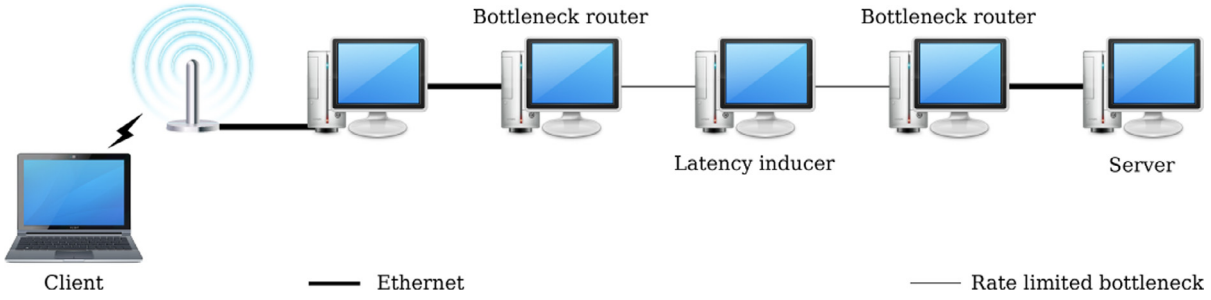


Fig. 8. WiFi test setup.

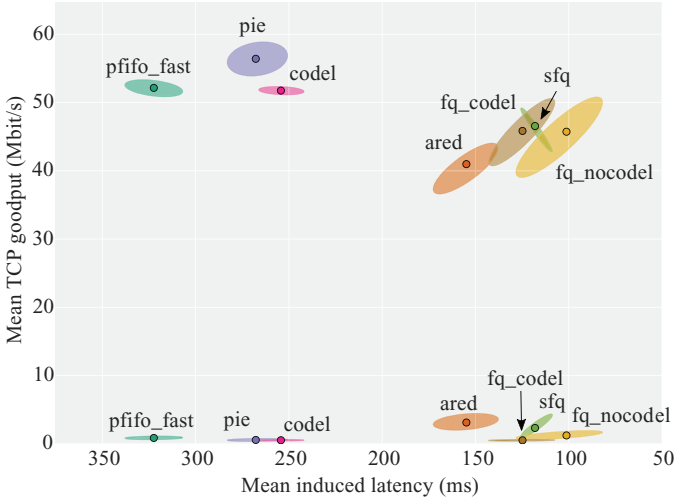


Fig. 9. RRUL results for the WiFi setup. The top part is downstream traffic, the bottom part upstream.

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

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.

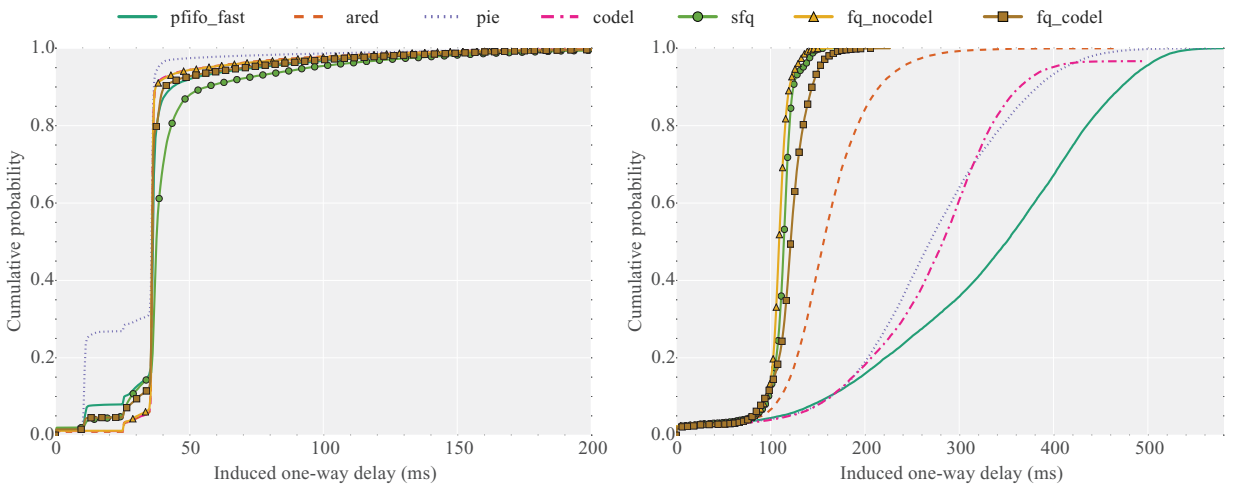
On this WiFi setup we have re-run all tests designed to test a single link characteristic, i.e. everything except the fairness test. However, for the lower bandwidths, the WiFi link does not constitute a bottleneck, and so we see no meaningful difference in the results.[4] For this reason, we have omitted 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 Fig. 9. A couple of interesting features are clearly visible on this graph. Firstly, the

[4] 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.



(a) 1 stream. **(b)** RRUL.

Fig. 10. VoIP test results for WiFi.

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) 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 Fig. 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.

7.3. Web results

The web results from the WiFi tests are shown in Fig. 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.

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

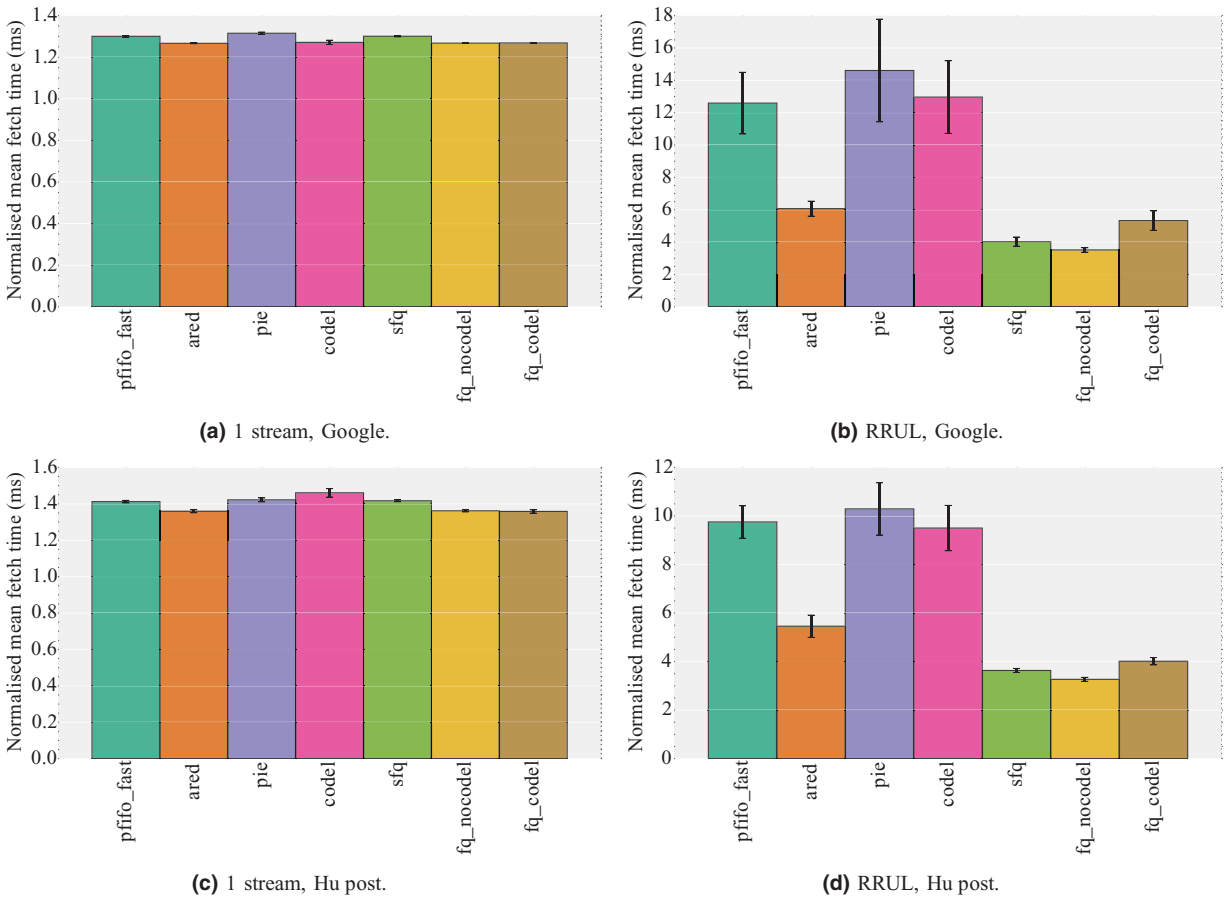
VoIP packet loss		
	1 stream (%)	RRUL (%)
pfifo_fast	0.34	16.66
ARED	0.13	5.30
PIE	0.18	27.52
CoDel	0.19	18.56
SFQ	0.47	1.71
fq_nocodel	0.17	1.59
fq_codel	0.22	2.64

7.4. 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 straightforward 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.



(a) 1 stream, Google.

(b) RRUL, Google.

(c) 1 stream, Hu post.

(d) RRUL, Hu post.

Fig. 11. Web test results for 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.

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 area 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,[5] 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.

Queue management surely plays an important role in ensuring tomorrow's Internet provides reliable 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 latenc at all layers.

References

[1] B. Briscoe, A. Brunstrom, A. Petlund, D. Hayes, D. Ros, I.-J. Tsang, S. Gjessing, G. Fairhurst, C. Griwodz, M. Welzl, Reducing internet latency: a survey of techniques and their merits, IEEE Commun. Surv. Tutorials PP (99) (2014) 1, doi:10.1109/COMST.2014.2375213.

[2] J. Gettys, K. Nichols, Bufferbloat: dark buffers in the internet, Queue 9 (11) (2011) 40:40–40:54, doi:10.1145/2063166.2071893.

[3] C. Staff, Bufferbloat: what's wrong with the internet? Commun. ACM 55 (2) (2012) 40–47, doi:10.1145/2076450.2076464.

[4] S. Floyd, V. Jacobson, Random early detection gateways for congestion avoidance, IEEE/ACM Trans. Networking 1 (4) (1993) 397–413.

[5] K. Nichols, V. Jacobson, Controlling queue delay, Commun. ACM 55 (7) (2012) 42–50, doi:10.1145/2209249.2209264.

[6] R. Pan, P. Natarajan, C. Piglione, M. Prabhu, V. Subramanian, F. Baker, B. VerSteeg, Pie: a lightweight control scheme to address the bufferbloat problem, in: 2013 IEEE 14th International Conference on High Performance Switching and Routing (HPSR), 2013, pp. 148–155, doi:10.1109/HPSR.2013.6602305.

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

[8] J. Gettys, Traditional aqm is not enough, Blog post, 2013. http://goo.gl/V6vajZ.

[9] P. McKenney, Stochastic fairness queueing, in: IEEE Proceedings of Ninth Annual Joint Conference of the IEEE Computer and Communication Societies. 'The Multiple Facets of Integration' (INFOCOM'90), vol. 2, 1990, pp. 733–740, doi:10.1109/INFCOM.1990.91316.

[10] T. Høiland-Jørgensen, P. McKenney, D. Taht, J. Gettys, E. Dumazet, FlowQueue-codel, Internet Draft (informational), 2014. http://tools.ietf.org/html/draft-ietf-aqm-fq-codel-00.

[11] R. Adams, Active queue management: a survey, IEEE Commun. Surv. Tutorials 15 (3) (2013) 1425–1476.

[12] F. Checconi, L. Rizzo, P. Valente, QFQ: Efficient packet scheduling with tight guarantees, IEEE/ACM Trans. Netw. (TON) 21 (3) (2013) 802–816 http://dl.acm.org/citation.cfm?id=2525552.

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

[14] W.-C. Feng, D. Kandlur, D. Saha, K. Shin, Stochastic fair blue: a queue management algorithm for enforcing fairness, in: IEEE Proceedings (INFOCOM 2001, IEEE, 2001, pp. 1520–1529.

[15] G. White, Active queue management in DOCSIS 3.X cable modems, Technical Report, 2014 http://www.cablelabs.com/wp-content/uploads/2014/06/DOCSIS-AQM_May2014.pdf.

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

[17] V.P. Rao, M.P. Tahiliani, 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, pp. 262–267.

[18] I. Järvinen, M. Kojo, Evaluating CoDel, PIE, and HRED AQM Techniques with Load Transients, IEEE, 2014.

[19] K. Cai, M. Blackstock, R. Lotun, M.J. Feeley, C. Krasic, J. Wang, Wireless unfairness: alleviate mac congestion first!, in: Proceedings of the Second ACM International Workshop on Wireless Network Testbeds, Experimental Evaluation and Characterization, in: (WinTECH'07, ACM, New York, NY, USA, 2007, pp. 43–50, doi:10.1145/1287767.1287777.

[20] G. Park, H. Ko, S. Pack, Simulation study of bufferbloat problem on wifi access point, in: 2014 IEEE 3rd Global Conference on Consumer Electronics (GCCE), 2014, pp. 729–730, doi:10.1109/GCCE.2014.7031276.

[21] M. Carbone, L. Rizzo, Dummynet revisited, ACM SIGCOMM Comput. Commun. Rev. 40 (2) (2010) 12–20.

[22] A.N. Kuznetsov, tbf—token ben bucket filer, Linux man page, 2014.

[23] D. Taht, J. Gettys, Best practices for benchmarking codel and fq codel, Wiki page on bufferbloat.net web site, 2014. http://goo.gl/FpSW5z.

[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, 2015, Open source benchmarking software. http://www.netperf.org/.

[26] A. Botta, A. Dainotti, A. Pescapè, A tool for the generation of realistic network workload for emerging networking scenarios, Comput. Netw. 56 (15) (2012) 3531–3547.

[27] D. Stenberg, Curl and libcurl, Project web site, 2015. http://curl.haxx.se/.

[28] T. Høiland-Jørgensen, Flent: the FLExible network tester, Project web site, 2015. https://flent.org.

[29] G. White, R. Pan, A PIE-based AQM for DOCSIS cable modems, Internet Draft (informational), 2015. http://tools.ietf.org/html/draft-white-aqm-docsis-pie-02.

[30] V. Misra, W.-B. Gong, D. Towsley, Fluid-based analysis of a network of AQM routers supporting TCP flows with an application to RED, SIGCOMM Comput. Commun. Rev. 30 (4) (2000) 151–160, doi:10.1145/347057.347421.

[31] D. Taht, Realtime response under load (rrul) test, 2012. https://github.com/dtaht/deBloat/blob/master/spec/rrule.doc?raw=true.

[32] K. Winstein, Transport architectures for an evolving Internet (Ph.D. thesis), Massachusetts Institute of Technology, 2014 http://web.mit.edu/keithw/www/Winstein-PhD-Thesis.pdf.

[33] H. Balakrishnan, V.N. Padmanabhan, How network asymmetry affects tcp, IEEE Commun. Mag. 39 (4) (2001) 60–67.

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

[35] A. Kortebi, L. Muscariello, S. Oueslati, J. Roberts, 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, V. Firoiu, D. Towsley, J. Kurose, Modeling tcp throughput: A simple model and its empirical validation, in: ACM SIGCOMM Computer Communication Review, vol. 28, ACM, 1998, pp. 303–314.

[5] 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.

[37] S. Ha, I. Rhee, L. Xu, Cubic: a new tcp-friendly high-speed tcp variant, SIGOPS Oper. Syst. Rev. 42 (5) (2008) 64–74, doi:10.1145/1400097.1400105.
[38] T. Kozu, Y. Akiyama, S. Yamaguchi, Improving rtt fairness on cubic tcp, in: 2013 First International Symposium on Computing and Networking (CANDAR), 2013, pp. 162–167, doi:10.1109/CANDAR.2013.30.
[39] R. Jain, D.-M. Chiu, W.R. Hawe, A Quantitative Measure of Fairness and Discrimination for Resource Allocation in Shared Computer System, Eastern Research Laboratory, Digital Equipment Corporation, 1984.
[40] B. Briscoe, [aqm] codel's control law that determines drop frequency, IETF AQM mailing list message, 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, 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, 2011. http://article.gmane.org/gmane.linux.network/213308/.

Toke Høiland-Jørgensen received his M.Sc. in Mathematics and Computer Science from Roskilde University, Denmark, in 2013 and is current a Ph.D. student at Karlstad University in Värmland, Sweden. His research interests include computer networking, with a special focus on reducing latency by controlling queues in the network. He has been involved in the bufferbloat community for more than two years and is the author of the Flent testing tool widely used in the community, as well as a contributor to the CeroWrt router firmware.

Per Hurtig received his Ph.D. in Computer Science in 2012 from Karlstad University, Sweden. In his thesis, he focused on low-latency transport for short-lived flows, a work that resulted in several mechanisms to reduce transport-level latency, some now available by default in the Linux kernel. Currently, he is assistant professor at the Department of Computer Science, Karlstad University where he mainly conducts research within the areas of transport-layer issues, congestion control, and network emulation. Per Hurtig has participated in a number of international EU-projects as well as national projects. He is also involved in Internet standardisation within the IETF.

Anna Brunstrom received a B.Sc. in Computer Science and Mathematics from Pepperdine University, CA, in 1991, and a M.Sc. and Ph.D. in Computer Science from College of William & Mary, VA, in 1993 and 1996, respectively. She joined the Department of Computer Science at Karlstad University, Sweden, in 1996, where she is currently a Full Professor and Research Manager for the Distributed Systems and Communications Group. Her research interests include transport protocol design, techniques for low latency Internet communication, cross-layer interactions, wireless communication and network security. She has authored/coauthored ten book chapters and over 100 international journal and conference papers.