The New AQM Kids on the Block: An Experimental Evaluation of CoDel and PIE

Naeem Khademi
Department of Informatics
University of Oslo
naeemk@ifi.uio.no

David Ros
Télécom Bretagne
Institut Mines-Télécom
david.ros@telecom-bretagne.eu

Michael Welzl
Department of Informatics
University of Oslo
michawe@ifi.uio.no

Abstract—**Active Queue Management (AQM) design has again come into the spotlight of network operators, vendors and OS developers. This reflects the growing concern and sensitivity about the end-to-end latency perceived by today's Internet users. CoDel and PIE are two AQM mechanisms that have recently been presented and discussed in the IRTF and the IETF as solutions for keeping latency low. To the best of our knowledge, they have so far only been evaluated or compared against each other using default parameter settings, which naturally presents a rather limited view of their operational range. We set thus to perform a broader experimental evaluation using real-world implementations in a wired testbed. We have in addition compared them with a decade-old variant of RED called Adaptive RED, which shares with CoDel and PIE the goal of "knob-free" operation. Surprisingly, in several instances results were favorable towards Adaptive RED.**

I. Introduction

End-to-end latency is one of the most important problems of the Internet today. Much awareness has been raised about this issue by the "Bufferbloat" project [16], [1], highlighting how common it is to experience buffer-induced latencies ranging from a few hundred milliseconds up to several seconds at the network's edge – e.g., in ADSL broadband modems, DOCSIS cable modems and 802.11 APs [12]. Ubiquitously deployed loss-based TCP congestion control mechanisms (e.g., standard SACK and Linux's CUBIC) tend to *persistently* fill any buffer; over-buffered devices can therefore worsen the user-perceived Internet performance, most specifically for latency-sensitive applications such as real-time interactive multimedia, online gaming and even web browsing, especially when they share the bottleneck with bulk TCP traffic.

Active Queue Management (AQM) is generally regarded as an important piece of the solution to this problem [21]. The primary goals of any AQM mechanism are: (a) to let the buffer absorb packet bursts that naturally occur in the networks while preventing it from building up standing queues; (b) to break any synchronization between flows. If possible, AQM mechanisms (refered to as "AQMs" hereafter) should also be able to protect flows from being starved by other more aggressive or misbehaving flows, as well as to support Explicit Congestion Notification (ECN).

While the history of AQM design begins around two decades ago with Random Early Detection (RED) by Floyd and Jacobson [14], it is only recently that the necessity of widespread AQM adoption has been fully realized by vendors and service providers. AQMs have occasionally been deployed in the Internet's core [2] or in ISPs' infrastructures [12], but in general there has been no extensive usage of AQMs in devices operating at the last hop of the access networks, although it has been long known that these devices often form the bottleneck of an end-to-end path. The lack of deployment of AQMs has often been attributed to the difficulty of correctly tuning the parameters of RED, the only mechanism that was described in an IETF Request for Comments [11] and the probably most commonly implemented one.

Recently, two new AQMs, CoDel [21] and PIE [24] have been proposed, along with a combination of CoDel with stochastic fair queuing (FQ_CoDel[1]) [15]. They promise to tackle the latency problem in lightly-to-moderately multiplexed environments. Thus, they are suitable candidates for implementation in devices operating on access links where a handful of flows might be sharing the bottleneck queue at any instant.

Despite claims of promising results achieved by these new AQMs, there is relatively little published work evaluating their performance. Exceptions are two (non peer-reviewed) preliminary studies from CableLabs [27], [26] which assess AQM performance on DOCSIS 3.0 modems and rely solely on ns-2 simulation results. Also, [17] conducts a preliminary investigation on the interaction of different AQMs and low-priority "scavenger" congestion control using real-life tests. In addition, the initial CoDel paper [21] includes a simulation study comparing CoDel with traditional RED. While [27], [17] study CoDel only, [26] also includes PIE and FQ_CoDel. Assertions about the performance of these AQM schemes on access links have not yet been fully substantiated by exhaustive real-life experiments.

The very recent work in [23] is close in style to our own evaluation study; however, it only uses default parameters, and the comparison with RED is based on simulations. In contrast, we investigate the effect of varying the parameters of the algorithms and compare against a real-life implementation of the auto-tuning variant of RED, ARED, which we consider to be a more appropriate algorithm to compare against.

Our work is meant to be a step towards a better understanding of these AQMs, with a focus on their important "knob-free" aspect, i.e. the impact of varying parameters in CoDel and PIE.

[1]FQ_CoDel created much attention in the research community. Its code is said to connect the CoDel and stochastic fair queuing (SFQ) algorithms in ways that go beyond merely applying CoDel per queue (hash bin) of SFQ. This behavior is not documented, making it hard to design good evaluation scenarios for FQ_CoDel. Clearly, the very different nature of scheduling and AQM algorithms warrant a much deeper investigation of this algorithm than what could be provided here, which is why FQ_CoDel is not discussed further in this paper.

A broader view of the performance of CoDel and PIE is given in an extended version of this paper, which is available as a technical report [19]; this report discusses many other topics and scenarios – e.g. 802.11 WLANs and operation with ECN. As one of the major contributions of the new mechanisms appears to be their parameterless operation, and given that an old "auto-tuning" variant of RED known as "Adaptive RED (ARED)" [13] exists, it is obvious to ask how well CoDel and PIE perform in comparison with ARED. We therefore try to answer this question too, both in the short and the extended version of this work.

The metrics under evaluation are the goodput and RTT experienced by bulk TCP transfers. This may superficially seem to be irrelevant due to the prevalence of short web-like transfers in today's Internet as well as the importance of bursty and unresponsive traffic that are typical of multimedia applications. While we believe that it is necessary to consider the performance of AQMs with such traffic and intend to do so in future work, it seems obvious to us that an investigation like the one presented here should start as simple as possible. Given that AQMs were typically designed with the dynamics of congestion control mechanism for bulk TCP transfers in mind, this traffic type seems to be the most natural choice as a starting point. Besides, bulk transfers do happen in the Internet today, and the effect of such transfers in terms of latency ("collateral damage") on other coexisting traffic such as interactive multimedia can be expected to resemble the effect experienced by the bulk TCP transfers themselves, especially when the interactive multimedia traffic is relatively small in volume (as it is typically the case for latency-critical applications, e.g. games or VoIP) [25].

The rest of this paper is organized as follows: Section II provides an overview of the AQMs studied in this paper. Section III presents the experimental setup used in the evaluations. An assessment of the chosen AQMs, in a variety of scenarios, is the subject of Section IV. Finally, Section V concludes the paper with the findings and future work.

II. AQMs For Low Latency

The most commonly mentioned reason for the lack of AQM deployment in today's Internet is the known difficulty in correctly setting the parameters of RED. It is therefore essential that a newly standardized[2] AQM requires very little parameter tuning, or, better yet, none at all.

Despite their authors' "no-knob" claim, both CoDel and PIE in fact maintain a set of parameters (with recommended default values) that do affect their performance. On the other hand, ARED adaptively sets RED's maximum drop probability parameter based on a desired target average queue. We can identify two key parameters in all three mechanisms:

1) **Target delay:** CoDel and PIE both maintain a value called $target\_delay$. However this parameter is used in different ways in these AQMs. While CoDel starts dropping packets when the queuing delay has been above $target\_delay$ for a certain amount of time, PIE continuously updates its dropping probability based on the difference between the current queuing delay and $target\_delay$. Although ARED does not explicitly maintain a $target\_delay$ value, when ARED is used where the bandwidth is fixed, it is possible to derive its corresponding $target\_queuing$ from a given target delay.
2) **Interval:** Most AQMs require a certain time interval to update their dropping/marking probability or decide whether to discard the incoming/outgoing packet(s). The semantics of this interval differs from one AQM to another. CoDel [21] uses the interval value to decide how long the queuing latency can stay above $target\_delay$ before switching to *dropping mode* (Section II-A). On the other hand, PIE and ARED use their interval to update the dropping/marking probability (Section II-B and Section II-C).

[2]The formation of a new IETF Working Group on AQM has been approved by the IESG on 26 September 2013.

Next, we provide a brief overview of the AQMs investigated in this paper. In addition to the two recent PIE and CoDel AQMs that adaptively try to keep the latency low, we have also included ARED both as a baseline for comparison and also because, to the best of our knowledge, ARED's performance has never been evaluated against the recent AQM mechanisms in the context of bufferbloat on access links.

A. CoDel

CoDel [21], [22] tries to detect a standing queue by tracking, using timestamps, the minimum queuing delay (or sojourn delay) packets experience in a fixed-duration interval, set to 100 ms by default. CoDel assumes that a small *target* standing queue is tolerable so as to achieve good link utilization. When the minimum queuing delay has exceeded the $target\_delay$ value during at least one interval, a packet is dropped from the tail and a control law is used to set the next dropping time. Using the well-known relationship of dropping rate to TCP throughput [20], this time interval is decreased in inverse proportion to the square root of the number of drops since the dropping state was entered, to ensure that TCP does not underutilize the link.

When the queuing delay goes below $target\_delay$, the controller stops dropping packets and exits the dropping state. In addition, no drops are carried out if the queue contains fewer than an MTU's worth of bytes. Additional logic avoids re-entering the dropping state too early after exiting it, and CoDel resumes the dropping state at a recent control level, if one exists. CoDel only enters the dropping state when the minimum queuing delay has exceeded $target\_delay$ for an interval long enough to absorb normal packet bursts (100 ms by default). This ensures that a burst of packets will not experience packet drops as long as the burst can be cleared from the queue within a reasonable period.

A variant of CoDel with stochastic fair queuing ("FQ_CoDel") has been available since Linux kernel version 3.5 to provide better fairness and flow isolation. Flow isolation in FQ_CoDel better protects flows from the impact of non-responsive flows such as constant bit-rate (CBR) multimedia traffic and also protects the thin streams e.g. interactive multimedia in presence of bulk TCP traffic.

TABLE I. PIE PARAMETERS IN [5].

PIE Parameter	Default value
t_{update}	30 ms
T_{target}	20 ms
α	2
β	20

B. PIE

The Proportional Integral controller Enhanced (PIE) AQM [24] randomly drops a packet at the onset of congestion. Similar to CoDel, it uses queuing latency instead of the more commonly used queue length. It uses the trend of latency over time (increasing or decreasing) to determine the congestion level.

Different from CoDel that drops packets on departure (dequeue time) and requires timestamping, PIE drops packets on arrival (enqueuing time) with probability p and does not require timestamping, making it a more lightweight mechanism. Every t_{update} time units, PIE estimates the current queuing delay using Little's law in Eq. 1 and derives p based on Eq. 2:

$$E[T] = N/\mu \tag{1}$$

$$p = p + \alpha * (E[T] - T_{target}) + \beta * (E[T] - E[T]_{old}) \tag{2}$$

where N is the current queue length, μ is the draining rate of the queue, $E[T]$ represents the current estimated queuing delay, $E[T]_{old}$ represents the previous iteration's estimation of the queuing delay, and T_{target} is the target queuing delay. The drop probability calculation incorporates the direction in which the delay is moving by employing a classic Proportional Integral (PI) controller design, similar to the one used in [18]. The α factor determines how the deviation of current latency from T_{target} affects the drop probability. The β factor makes additional adjustments depending on whether the latency trend is positive or negative. The default values of PIE parameters based on the Linux implementation in [5] are shown in Table I.

C. ARED

One of the main historical challenges with the deployment of RED [14] has been the tuning of its parameters, so that the average queue length stays approximately around a desired $target\_queuing$ value[3] in the presence of different levels of congestion, when RED's performance is hard to estimate in advance. This has been a discouraging factor for network operators to deploy RED on congested routers, where predictable queuing latency is important. Adaptive RED (ARED) [13] tries to solve this problem by dynamically adjusting RED's maximum drop probability (p_{max}). ARED observes the average queue length ($\bar{N}$) to infer whether to make RED more/less aggressive.

Similar to RED, ARED keeps two thresholds (th_min and th_max) which, to correlate with a single $target\_queuing$ value, are set to $0.5 * target\_queuing$ and $1.5 * target\_queuing$ in accordance with the rules in [13]. If $\bar{N}$ moves towards th_min, early detection is too aggressive. On the other hand, if $\bar{N}$ moves towards th_max, early detection is too conservative. Using an Additive Increase / Multiplicative Decrease (AIMD) policy (see Table II), ARED adaptively changes p_{max} so that the average queue length oscillates around $(th\_max+th\_min)/2$. ARED updates p_{max} periodically after every $interval$ (500 ms by default).

[3]Bear in mind that $target\_queuing$ can correspond to a certain $target\_delay$ in fixed bandwidth scenarios.

TABLE II. ARED PARAMETERS IN [13].

ARED Parameter	Default value
$interval$	500 ms
α	$min(0.01, p_{max}/4)$
β	0.9

Although ARED [13] was proposed almost a decade and half ago, along with plenty of other RED variants (surveyed in [7]) and even though it has been available since Linux kernel version 3.3, to the best of our knowledge, its real-life performance has not been investigated yet.

III. PERFORMANCE EVALUATION SETUP

While the "older generation" of AQMs were designed without any specific distinction on where in the network they can be deployed, the two new AQMs (CoDel and PIE) are designed to overcome the bufferbloat problem on access links. It is therefore vital to investigate the deployment of AQMs where they belong. For instance, an AQM designed for access link scenarios cannot achieve buffering latency requirements that are common in data centers. The experimental network testbed setup that we used in this paper is therefore based on the assumption of AQM deployment on access links.

A. Experimental Setup

We used a wired dumbbell topology in our real-life experiments. A detailed summary of the hardware used in these experiments is given in Table III.

1) Wired Topology: Our wired testbed setup consists of a dumbbell network topology with two sets of Dell OptiPlex GX620 machines acting as senders and receivers in a dumbbell topology. The senders and receivers are connected via two routers; the first router is acting as an *AQM router* implementing AQM on its bottleneck (egress) interface. All senders are connected to the AQM router's ingress interface using a switched network with 100 Mbps Ethernet links. The second router is acting as a delay node on both forward and reverse traffic, providing an RTT_{base} for the traffic traversing it using the $ipfw$ dummynet module [4]. This router is connected to the receivers with switched 100 Mbps links. The egress interface of the AQM router (bottleneck interface) and the ingress interface of the second router are set to 10 Mbps advertised mode using $ethtool$ [3] with auto-negotiation being on to establish the bottleneck link.

2) Generic Setup: All nodes are synchronized using *ntp*. Each experiment consists of 60 sec of $iperf$ traffic between each sender/receiver pair. Each run is repeated 10 times and the presented results are the averages of all runs. RTT_{base} is set to 100 ms by default. TCP-Segmentation-Offload (TSO) engine is turned off by default on all nodes' NICs. The bottleneck's maximum queue size ($txqueuelen$) is set to 1000 packets and its Byte-Queue-Limit (BQL) value is set to 1514 bytes.

TABLE III. EXPERIMENTAL NETWORK TESTBED SETUP

Model	Dell OptiPlex GX620
CPU	Intel(R) Pentium(R) 4 CPU 3.00 GHz
RAM	1 GB PC2-4200 (533 MHz)
Ethernet	Broadcom NetXtreme BCM5751 RTL-8139 (*AQM interface*) RTL8111/8168B (*Dummynet* router)
Ethernet driver	tg3 8139too (*AQM* interface) r8168 (*Dummynet* router)
802.11 b/g	D-Link DWL-G520 AR5001X+
802.11 driver	ath5k
OS kernel	Linux 3.8.2 (FC14) Linux 3.10.4 (*AQM* router) (FC16)

TCP's appropriate-byte-counting (ABC) [8], SACK [10] and D-SACK options [9] are enabled. TCP's transmit and receiver buffers are set large enough to be able fully utilize bandwidth×delay ($C.RTT_{base}$) on the bottleneck path and the congestion control algorithm is set to SACK (named *reno* in Linux). We used the Hierarchical Token Bucket (HTB) Linux queuing discipline to set the AQM on the bottleneck interface. The AQM *qdisc* size is also set equal to *txqueuelen* (1000 packets) for all experiment runs.

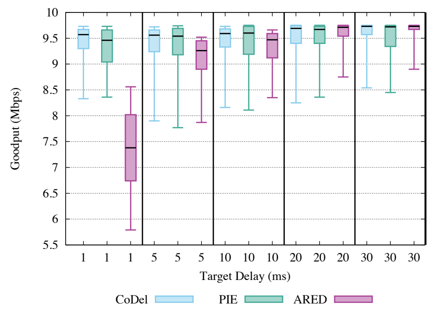
All traffic is dumped using *tcpdump* on all interfaces in the network. We use the Synthetic Packet Pairs (SPP) tool [6] to precisely measure the actual per-packet RTT perceived on the path. To measure the TCP throughput/goodput, *tcptrace* is used. CoDel and ARED's modules are available by default in Linux 3.10.4 kernel while we used the PIE module code available in [5] for our experiments. All AQMs use their default values unless otherwise noted. In case of ARED, the parameters are calculated based on the recommendations in [13] except when *target_delay*=1 ms where *th_min* and *th_max* are set to 1 MTU and 2 MTUs accordingly.

IV. Evaluating AQM Designs

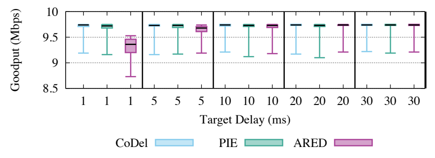
The AQMs under consideration in this paper have two key parameters, and their tuning affects their performance under different network conditions (see Section II). We investigate how these AQMs perform when varying their parameter settings and the level of congestion in the network. While the notion of *target_delay* and *interval* differ between the two algorithms, the trade-off controlled by the parameters is similar, and hence we group them together: *target_delay* is concerned with directly limiting the delay experienced by the traffic at the potential cost of utilization, and *interval* is concerned with allowing bursts of a certain magnitude at the potential cost of increased delay.

A. Target Delay

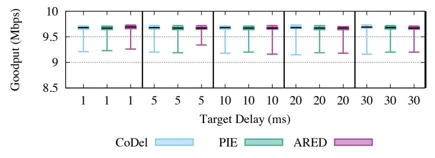
Figure 1 shows the per-packet RTT distributions of CoDel, PIE and Adaptive RED for lightly, moderately and heavily congested link scenarios respectively, and for various *target_delay* values ranging from 1 to 30 ms (such range includes PIE's 20 ms and CoDel's 5 ms defaults). We define *lightly*, *moderately* and *heavily* congested link scenarios as when 4, 16 and 64 TCP flows associated with 4 sender/receiver pairs are sharing the bottleneck, respectively (i.e., 1, 4 or 16 flows per sender-receiver pair). Since RTT_{base} is set to 100 ms, the difference between values on whisker-box plots and 100 ms correspond to queuing delay at the bottleneck. We refer to this as "queuing delay" hereafter.



(a) Light



(b) Moderate



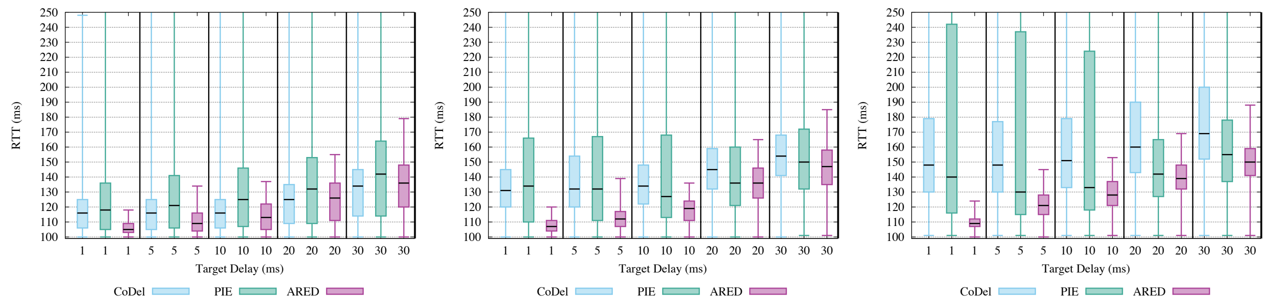
(c) Heavy

Fig. 2. TCP goodput at the bottleneck link (per 5-sec intervals). Light, Moderate and Heavy congestion scenarios (with 4 senders and RTT_{base}=100 ms); bottom and top of whisker-box plots show 10th and 90th percentiles, respectively.

Here we can make several observations: CoDel and PIE's median, 10th and 90th percentiles of queuing delay increase proportionally to the level of congestion at the bottleneck, whereas Adaptive RED better controls the median latency at closer levels to *target_delay* for all extents of congestion. While Adaptive RED's median and 10th/90th percentiles of queuing delay decrease proportionally to the decrease of *target_delay*, PIE and CoDel's median and 10th/90th percentiles don't exhibit a close correlation with the choice of *target_delay* more specifically for higher congestion levels.

It is also observable that in general, and more clearly under high congestion, PIE and CoDel show longer queuing delay distribution tails (the difference between 10th and 90th percentiles), meaning more fluctuations in RTTs. CoDel's distribution tail length stays roughly equal for different target values within each congestion level scenario, while PIE's distribution tail tend to increase with decrease in target delay when the link becomes more congested. PIE in particular yields very large fluctuations in delay under load, for small values of *target_delay* ($\leq$ 10 ms) (see Figure 1(c)). Adaptive RED, on the other hand, exhibits much shorter distribution tails (which get shorter with decreasing *target_delay*), meaning more controlled queuing delay.

This behaviour is connected to the dropping function used in ARED. For lower values of *target_delay* the gap between ARED's derived *th_min* and *th_max* becomes very small. Since ARED's maximum drop probability (similar to RED)



(a) Light (b) Moderate (c) Heavy

Fig. 1. Per-packet RTT. Light, moderate and heavy congestion scenarios (4 senders and RTT_{base}=100 ms); bottom and top of whisker-box plots show 10th and 90th percentiles, respectively.

increases to 1 whenever the average queue length grows beyond th_max, ARED exhibits a very aggressive dropping behavior that keeps the latency distribution tail tight when $target\_delay$ is low. This leads to a better latency control for bulk TCP transfers while harming bursty traffic that might coexist with such traffic. The study of bursty traffic is out of this paper's scope and regarded as future work.

Figure 2 shows the achieved goodput of AQMs for different congestion levels and target delays related to the scenarios in Figures 1(a), 1(b) and 1(c). In the lightly congested scenario, Adaptive RED's median goodput drops to 7.38 Mbps when target delay is set to the extremely low value of 1 ms, otherwise it always stays between 9.26 and 9.73 Mbps for target delays from 5 to 30 ms. This is close to PIE and CoDel's median goodput which are around 9.46$\sim$9.72 Mbps and 9.57$\sim$9.73 Mbps for all target delay values respectively. Similarly, in the moderately congested scenario, Adaptive RED achieves almost exactly the same goodput as PIE and CoDel for all values of target delay except for the extreme low target of 1 ms. This similarity is even more obvious in the highly congested scenario where all AQMs yield $\approx$9.7 Mbps as the maximum achievable goodput on the bottleneck link.

B. Interval Time

CoDel and PIE use an update interval time that can be adjusted by the operator at user-space. ARED however uses a static fixed interval time of 500 ms, and therefore we only study CoDel and PIE in this part. As explained in Sections II-A and II-B, CoDel enters its dropping state if the minimum queuing delay exceeds target delay for a duration of one $interval$ while PIE uses an interval (t_{update}) to estimate the current queuing delay and update the drop probability. Despite the basic difference between CoDel and PIE's interval semantics, the parameter controls the magnitude of acceptable bursts in both algorithms. We consider three time-granularities at which AQMs might perform: fine (5 ms), medium (30 ms) and coarse (100 ms) relative to the RTT_{base} of 100 ms. This set of values incorporates CoDel's (100 ms) and PIE's (30 ms) default interval values as well.

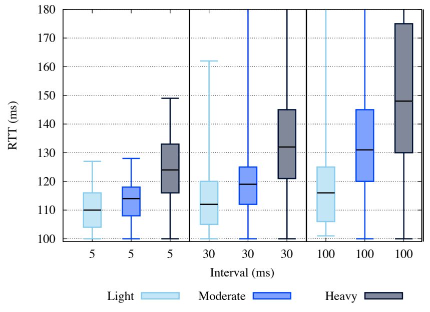
a) CoDel's dropping-mode interval: Figure 3 shows CoDel's performance when its dropping-mode interval is set to the above values and for different levels of congestion. While this parameter obviously controls the trade-off between delay and goodput, the figure indicates that a smaller value than the default one could be recommendable: for RTT_{base}=100 ms, using a smaller interval than the default 100 ms leads to significant improvement in median queuing delay (37.5%$\sim$54.8%) while compromising 3.56% of median goodput only when the congestion level is light (Figure 3(b)). This confirms our belief that CoDel's default 100 ms interval is set unnecessarily high since it admits packet bursts equal to 100 ms worth of data transfer without any drop when entering the dropping mode for the first time. The rationale behind this "magic number" relates to the assumption made in [21] that 100 ms is the average RTT on the today's Internet and CoDel should be able to allow a maximum burst size equal to $C.RTT_{base}$ worth of data when a single flow is present at the bottleneck. This assumption is however not relevant for scenarios with multiple flows and also when RTT_{base} can differ by several orders of magnitude.

b) PIE's t_{update} interval: Figure 4 shows PIE's performance for different values of t_{update}. As expected, PIE achieves better queuing delays with fine-grained intervals and better goodput with coarse-grained intervals. The queuing delay's trend is more obvious in the lightly congested scenario but becomes less predictable as the congestion level increases (Figure 4(a)). In the lightly congested scenario, PIE's median queuing latency can be improved by 28.6% by reducing the default t_{update} from 30 ms to 5 ms while compromising only 1.86% of median goodput (Figure 4(b)).

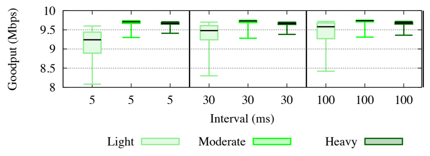
V. Concluding Remarks

Our investigation on the impact of parameter settings in CoDel and PIE lets us derive a clear recommendation for CoDel: its dropping mode interval is set too large by default, as reducing it seems to lower the delay without degrading the goodput much. Somewhat surprisingly, PIE, as implemented in Linux, yielded a long distribution tail for low target delays in our tests. We were also surprised to find that ARED *only* performed worse than CoDel and PIE when the number of flows on the bottleneck link was very small; its more aggressive behavior above th_max and tight thresholds for lower target delays gave it a tight and low delay distribution. We therefore consider it worthwhile to further investigate ARED in future work.

As explained in the outset, we consider our work as a first fundamental step towards a better understanding of the performance of CoDel and PIE. By its nature, such a first step has its limitations, and so this paper only focused on bulk TCP transfers. Our plans for future work therefore also includes more realistic traffic types (including bursty traffic,

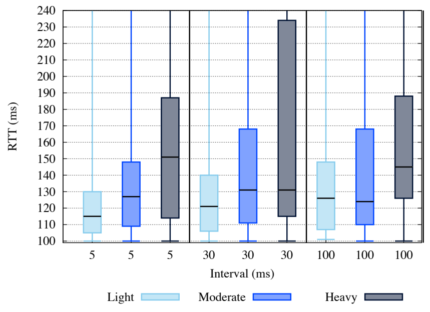


(a) Per-packet RTT

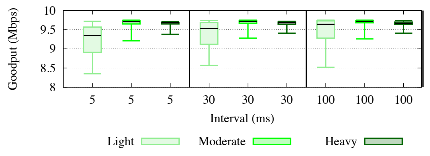


(b) Goodput at the bottleneck

Fig. 3. Per-packet RTT and bottleneck's per 5-sec TCP goodput. Varying CoDel's dropping-mode interval (4 senders, RTT_{base}=100 ms and $target\_delay$=5 ms); bottom and top of whisker-box plots show 10th and 90th percentiles, respectively.



(a) Per-packet RTT



(b) Goodput at the bottleneck

Fig. 4. Per-packet RTT and bottleneck's per 5-sec TCP goodput. Varying PIE's t_{update} interval (4 senders, RTT_{base}=100 ms and $target\_delay$=5 ms); bottom and top of whisker-box plots show 10th and 90th percentiles, respectively.

to test AQMs for their burst absorption capability), as well as simulations to stretch environment parameters to conditions that cannot be produced with our testbed.

VI. Acknowledgement

This work was partly funded by the European Community under its 7th Framework Programme through the Reducing Internet Transport Latency (RITE) project (ICT-317700) and the Aurora–Hubert Curien Partnership project "ANT" (grant number 28844PD/221629). The views expressed are solely those of the authors.

References

[1] Bufferbloat. http://www.bufferbloat.net/.

[2] Cisco WRED Guide. http://www.cisco.com/en/US/docs/ios/qos/configuration/guide/config_wred.pdf.

[3] Ethtool. http://linux.die.net/man/8/ethtool.

[4] IPFW. http://info.iet.unipi.it/~luigi/dummynet/.

[5] PIE Linux code (Cisco). ftp://ftpeng.cisco.com/pie/linux_code/.

[6] Synthetic Packet Pairs. http://caia.swin.edu.au/tools/spp/.

[7] R. Adams. Active Queue Management: A Survey. *IEEE Communications Surveys Tutorials*, 15(3):1425–1476, 2013.

[8] M. Allman. TCP Congestion Control with Appropriate Byte Counting (ABC). RFC 3465 (Experimental), Feb. 2003.

[9] E. Blanton and M. Allman. Using TCP Duplicate Selective Acknowledgement (DSACKs) and Stream Control Transmission Protocol (SCTP) Duplicate Transmission Sequence Numbers (TSNs) to Detect Spurious Retransmissions. RFC 3708 (Experimental), Feb. 2004.

[10] E. Blanton, M. Allman, L. Wang, I. Jarvinen, M. Kojo, and Y. Nishida. A Conservative Loss Recovery Algorithm Based on Selective Acknowledgment (SACK) for TCP. RFC 6675 (Proposed Standard), Aug. 2012.

[11] B. Braden, D. Clark, J. Crowcroft, B. Davie, S. Deering, D. Estrin, S. Floyd, V. Jacobson, G. Minshall, C. Partridge, L. Peterson, K. Ramakrishnan, S. Shenker, J. Wroclawski, and L. Zhang. Recommendations on Queue Management and Congestion Avoidance in the Internet. RFC 2309 (Informational), Apr. 1998.

[12] M. Dischinger, A. Haeberlen, K. P. Gummadi, and S. Saroiu. Characterizing Residential Broadband Networks. In *ACM SIGCOMM Conference on Internet Measurement (IMC)*, IMC, San Diego, CA, USA, Oct. 2007.

[13] S. Floyd, R. Gummadi, and S. Shenker. Adaptive RED: An Algorithm for Increasing the Robustness of RED's Active Queue Management. Technical report, 2001.

[14] S. Floyd and V. Jacobson. Random Early Detection Gateways for Congestion Avoidance. *IEEE/ACM Trans. Netw.*, 1(4):397–413, Aug. 1993.

[15] J. Gettys. fq_codel Status. Presentation at the 87th IETF meeting.

[16] J. Gettys. Bufferbloat: Dark Buffers in the Internet. *IEEE Internet Computing*, 15(3):96, 2011.

[17] Y. Gong, D. Rossi, C. Testa, S. Valenti, and D. Taht. Fighting the Bufferbloat: On the Coexistence of AQM and Low Priority Congestion Control. In *IEEE INFOCOM Workshop on Traffic Monitoring and Analysis (TMA)*, Apr. 2013.

[18] C. Hollot, V. Misra, D. Towsley, and W.-B. Gong. On Designing Improved Controllers for AQM Routers Supporting TCP Flows. In *IEEE INFOCOM*, Anchorage, AK, USA, Apr. 2001.

[19] N. Khademi, D. Ros, and M. Welzl. The New AQM Kids on the Block: Much Ado About Nothing? *Research Report 434*, 2013.

[20] M. Mathis, J. Semke, J. Mahdavi, and T. Ott. The Macroscopic Behavior of the TCP Congestion Avoidance Algorithm. *SIGCOMM Comput. Commun. Rev.*, 27(3):67–82, July 1997.

[21] K. Nichols and V. Jacobson. Controlling Queue Delay. *ACM Queue*, 10(5):20:20–20:34, May 2012.

[22] K. Nichols and V. Jacobson. Controlled Delay Active Queue Management. Internet-Draft draft-nichols-tsvwg-codel-01.txt, Feb. 2013.

[23] R. Pan, P. Natarajan, C. Piglione, M. Prabhu, V. Subramanian, F. Baker, and B. VerSteeg. PIE: A Lightweight Control Scheme to Address the Bufferbloat Problem. In *IEEE Conference on High Performance Switching and Routing (HPSR)*, July 2013.

[24] R. Pang. PIE: A Lightweight Control Scheme to Address the Bufferbloat Problem. Internet-Draft draft-pan-tsvwg-pie.txt, June 2013.

[25] L. Stewart, D. A. Hayes, G. Armitage, M. Welzl, and A. Petlund. Multimedia-unfriendly TCP Congestion Control and Home Gateway Queue Management. In *ACM Conference on Multimedia Systems (MMSys)*, San Jose, CA, USA, Feb. 2011.

[26] G. White. A Simulation Study of CoDel, SFQ-CoDel and PIE in DOCSIS 3.0 Networks. Technical Report, CableLabs, Apr. 2013.

[27] G. White and J. Padden. Preliminary Study of CoDel AQM in a DOCSIS Network. Technical Report, CableLabs, Nov. 2012.