PI^2: A Linearized AQM for both Classic and Scalable TCP

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

ABSTRACT

This paper concerns the use of Active Queue Management (AQM) to reduce queuing delay. It offers insight into why it has proved hard for a Proportional Integral (PI) controller to remain both responsive and stable while controlling 'Classic' TCP flows, such as TCP Reno and Cubic. Due to their non-linearity, the controller's adjustments have to be smaller when the target drop probability is lower. The PI Enhanced (PIE) algorithm attempts to solve this problem by scaling down the adjustments of the controller using a look-up table. Instead, we control an internal variable that is by definition linearly proportional to the load, then post-process it into the required Classic drop probability—in fact we show that the output simply needs to be squared. This allows tighter control, giving responsiveness and stability better or no worse than PIE achieves, but without all its corrective heuristics.

With suitable packet classification, it becomes simple to extend this PI^2 AQM to support coexistence between Classic and Scalable congestion controls in the public Internet. A Scalable congestion control ensures sufficient feedback at any flow rate, an example being Data Centre TCP (DCTCP). A Scalable control is linear, so we can use the internal variable directly without any squaring, by omitting the post-processing stage.

We implemented PI^2 as a Linux qdisc to extensively test our claims using Classic and Scalable TCPs.

1. INTRODUCTION

Interactive latency-sensitive applications are becoming prevalent on the public Internet, e.g. Web, voice, conversational and interactive video, finance apps, online gaming, cloud-based apps, remote desktop. It has been conjectured that there is also latent demand for more interactivity and new interactive apps would surface if there were less lag in the public Internet [10]. In the developed world, increases in access network bitrate have been giving diminishing returns as latency has become the critical bottleneck. Recently, much has been done to reduce propagation delay, e.g. by placing caches or servers closer to users. However, latency is a multi-faceted problem [7], so other sources of delay such as queuing have become the bottleneck.

Queuing delay problems occur when a capacity-seeking (e.g. TCP) traffic flow is large enough to last long enough to build a queue in the same bottleneck as traffic from a delay-sensitive application. Therefore queuing delay only appears as an intermittent problem [17]. Nonetheless, perception of quality tends to be dominated by worst case delays and many real-time apps adapt their buffering to absorb all but worst-case delays.

To remove unnecessary queuing delays, Active Queue Management (AQM) is being deployed at bottlenecks. AQM introduces a low level of loss, to keep the queue shallow within the buffer. However, for an AQM to reduce queuing delay any further, it has to worsen drop and/or utilization. This is because the root cause of the problem is the behaviour of current TCP-friendly congestion controls (Reno, Cubic, etc.). They behave like a balloon; if the network squeezes one impairment, they make the others bulge out.

To remove one dimension from this trilemma, it would seem Explicit Congestion Notification (ECN) could be used instead of loss. However, the meaning of an ECN mark was standardized as equivalent to a loss [30], so although standard ('Classic') ECN can remove the impairment of loss itself, it cannot be used to reduce queuing delay relative to loss-based protocols.

Per-flow queuing has been used to isolate each flow from the impairments of others, but this adds a new dimension to the trilemma; the need for the network to inspect within the IP layer to identify flows, not to mention the extra complexity of multiple queues.

[*]The first two authors contributed equally

CoNEXT '16, December 12 - 15, 2016, Irvine, CA, USA
© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM. ISBN 978-1-4503-4292-6/16/12…$15.00
DOI: http://dx.doi.org/10.1145/2999572.2999578

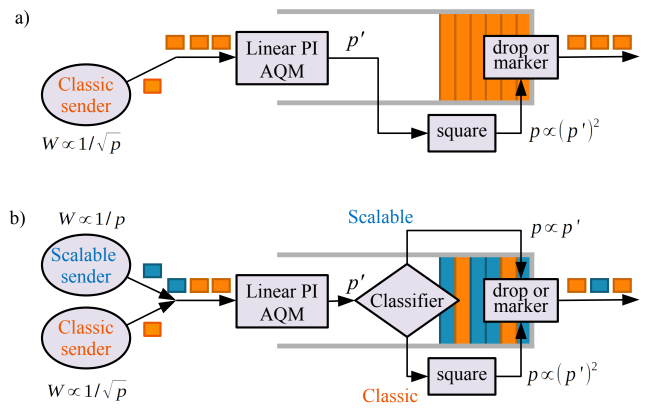

Figure 1: Controlling the window W with drop/marking probability p. a) Linearized Replacement for PIE; b) Congestion Control Coexistence

To reduce queuing delay below that achievable with state-of-the-art AQMs, and without other compromises, we decided to investigate changing TCP itself. Data Centre TCP (DCTCP [2]) is an existence proof of this approach; drastically reducing queuing delay without compromising the other factors. We shall show later that this is because, as flow-rate scales, the sawtooth variations in rate grow with TCP-friendly controls, but they do not with DCTCP, which can keep queuing delay low without compromising utilization. Thus DCTCP is an example of a 'Scalable' congestion control. In contrast, we use the term 'Classic' for unscalable controls such as Reno and Cubic.

It is a common misconception that DCTCP only works in data centres but, on its own, it works fine over wide area networks [3, 23]. However, it needs some additions for safety, which are discussed in [8]. The most critical problem is that DCTCP is too aggressive to coexist with Classic TCP. Therefore (until now) DCTCP could only be deployed in controlled environments like data centres because senders, receivers and the network all have to be updated at the same time.

During our research to make the family of Scalable congestion controls incrementally deployable on the Internet (see DualQ concepts in [13, 5]), we were interested in reusing the PIE (PI Enhanced [28]) AQM, as it can control target queuing delay and provide a reasonably stable marking probability. We modified it to separately support both Classic and Scalable congestion controls. This led us to understand that a proportional integral (PI) controller is inherently linear when controlling Scalable congestion controls, but not Classic ones.

The designers of PIE introduced scaling factors within the controller that increase stepwise as the drop/mark probability p increases, in order to keep it within its stable region. We realized that it would be much simpler to break down the structure of the AQM into two parts (Figure 1a): i) a generic part that controls a pseudo-probability p' that is, by definition, linearly proportional to the load; and ii) a congestion-control-specific part, which encodes p' into a congestion signal, p, appropriate to the congestion control that is predominant in the network.

For the case of Classic controls like TCP Reno, we shall see that the second stage should be configured to encode p as the square of p', which counterbalances the non-linear square root in the equations for the window (W) of Classic congestion controls (Figure 1a). We call this 'PI Improved with a square' or PI^2. We shall show that PI^2 can achieve similar or sometimes even better results than PIE, while using a simpler algorithm.

For Scalable controls like DCTCP, the output of the PI controller (p') can be used directly for congestion marking; no encoding is needed. This makes it feasible to solve the problem of coexistence of Scalable and Classic traffic. The PI controller can be used for the former and PI^2 for the latter (Figure 1b). This counterbalances the more aggressive response of scalable algorithms with more aggressive congestion notification (given $0 \leq p' \leq 1$, it is greater without the square).

In this paper we hold back from discussing the DualQ concept, and instead focus on restructuring PIE in single queue scenarios. The majority of the paper is about simply replacing PIE with PI^2 for Classic traffic. Nonetheless, we also test how well PI^2 would support coexistence of Scalable and Classic congestion controls.

We must emphasize that a single queue is neither the optimal nor the recommended deployment arrangement for coexistence between Scalable and Classic traffic, because the Scalable traffic suffers the high queuing delay of the Classic traffic. However, it provides interesting insights and a basis for future development of DualQ variants [12].

The paper is organized as follows. Section 2 gives background on the scalability of congestion controls and the PI AQM controller. Section 3 provides insights into related work particularly the evolution of the Proportional Integral controller. Section 4 describes how PI can be improved with a square, while Section 5 provides details of the PI^2 AQM implementation. Experimental evaluation of PI^2 in comparison with PIE is presented in Section 6. Conclusions are summarized in Section 7. Appendices A and B respectively provide steady-state throughput equations for DCTCP and a stability analysis for the PI^2 AQM.

2. BACKGROUND

Congestion Control Scaling.

A congestion control like Reno or Cubic is termed unscalable because, as flow rate scales, the number of round trips between congestion signals increases. This leads to less tight control and higher variation of queuing and/or utilization.

The number of congestion signals per round trip in the steady state is

$$c = pW, \tag{1}$$

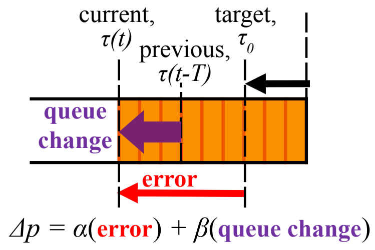

Figure 2: Basic PI algorithm

where W is the number of segments per round trip (the window) and p is the probability of congestion notification (either losses or ECN marks). Appendix A gives the formulae for the steady-state window of various congestion controls, which are all of the form

$$W \propto 1/p^B, \quad (2)$$

where B is a characteristic constant of the congestion control.

The scalability of a congestion control can therefore be determined by substituting for p from (2) into (1):

$$c \propto W^{(1-1/B)}. \quad (3)$$

If $B < 1$, c shrinks as W scales up, which implies fewer congestion signals per round trip (unscalable). Therefore, a congestion control is scalable if $B \geq 1$ and unscalable otherwise.

From Appendix A, Reno and Cubic in its Reno mode ('CReno') have $B = 1/2$ and pure Cubic has $B = 3/4$, so all these 'Classic' controls are unscalable. While DCTCP with probabilistic marking has $B = 1$ and with step marking it has $B = 2$. Therefore DCTCP is scalable irrespective of the type of marking.

Proportional Integral (PI) AQM.

The core of the PIE AQM is a classical Proportional Integral (PI) controller [18] that aims to keep queuing delay to a target τ_0 by updating the drop probability, p, every update interval, T. Figure 2 recaps the basic PI algorithm, which consists of a proportional and an integral part, weighted respectively by the gain factors β and α (both in units of Hz):

$$p(t) = p(t-T) + \alpha\big(\tau(t) - \tau_0\big) + \beta\big(\tau(t) - \tau(t-T)\big), \quad (4)$$

where $\tau(t)$ is the queuing delay at time t. The proportional part estimates how much load exceeds capacity by measuring how fast the queue is growing. The integral part corrects any standing queue due to the load having exceeded capacity over time, by measuring the error between the actual queuing delay and the target. Either term can be negative and the final probability is bounded between and including 0 and 1.

The proportional gain factor β can be seen as the best known correction of p to reduce the excess load. The integral gain factor α is typically smaller than the proportional, and corrects any longer term offset from the target. Note that the integral part initially calculates a proportional value, and the proportional part a differential value. But these values are later integrated by adding them as a delta to the probability used in the previous update interval.

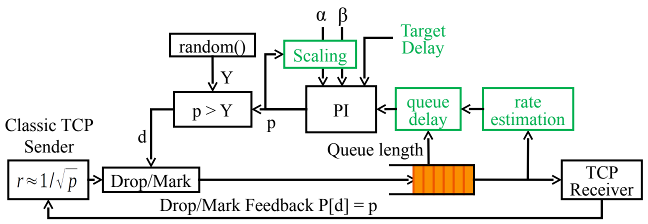

Figure 3: PIE AQM in Linux (additions to PI in green)

3. RELATED WORK

The evolution of the Proportional Integral controller Enhanced (PIE) AQM started with the control theoretic analysis of RED by Holot *et al* [19], which ended by pointing out that it was not necessary to adopt the approach of RED, which pushes back against higher load with higher queuing delay and higher loss. Instead, in [18], the same authors presented a Proportional Integral (PI) controller, using classical linear systems analysis with the objective of holding queuing delay to a constant target, using higher loss alone to push back against higher load.

This PI controller was used as the basis of the IEEE Data Centre Bridging (DCB) standard, 802.1Qau [15], published in 2010. Variants of the original PI AQM controller had also been proposed in the research community. In 2004, Hong *et al* claimed that the phase margin of the original PI controller could be over-conservative, leading to unnecessarily sluggish behaviour. Instead, they proposed a design [21] that would self-tune to the specified phase margin. In 2007, Hong and Yang [20] proposed to self-tune the gain margin instead, and to trigger the self-tuning process whenever it moved outside an operating range. However implementations have not adopted these self-tuning designs, probably because they require code to estimate the average (harmonic mean) round trip time of the TCP flows, which in turn depends on estimating the number of TCP flows, the link capacity and the equilibrium dropping probability, p. Of course, p is itself the output of the controller and, by definition, determining its equilibrium value is problematic when network conditions are changing, which is when self-tuning is required.

In 2013, the Proportional Integral controller Enhanced (PIE [28]) was proposed as an evolution of the original PI algorithm [18]. After extensive evaluation [32], PIE became the mandatory AQM algorithm for DOCSIS3.1 cable modems [9] and it is also being documented in an IETF specification [29].

Figure 3 shows the building blocks of the Linux PIE AQM implementation. PIE introduced three enhancements over PI. The first was to use units of time (not bytes or packets) for the queue, to hold queuing delay constant whatever the link rate. This ensures the

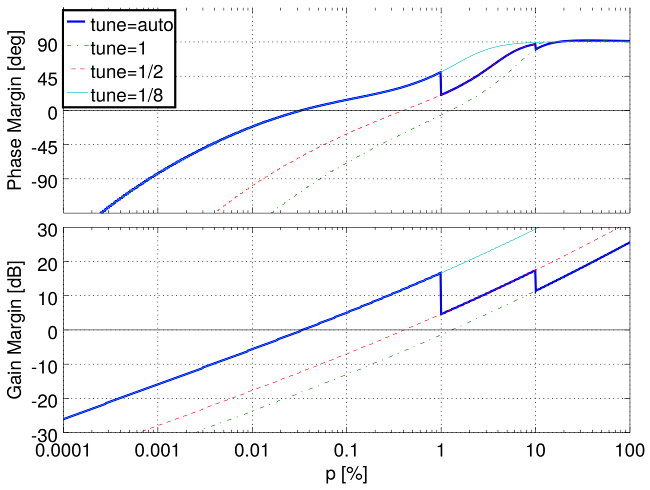

Figure 4: Bode plot margins for R=100 ms, α_{PIE}=0.125*tune, β_{PIE}=1.25*tune, T=32 ms

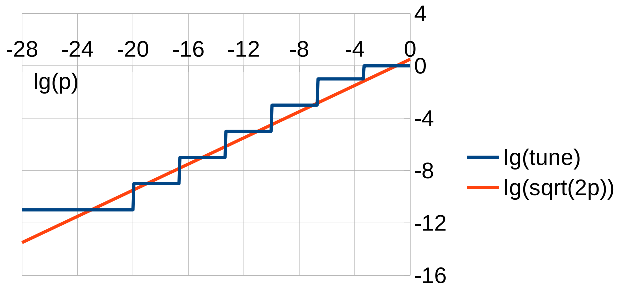

Figure 5: The factor 'tune' (stepped) that PIE uses to scale Δp from the lookup table in [29], compared against $\sqrt{2p}$ (both log scale, base 2)

algorithm does not need configuring for different link rates, which could occur due to modem retraining, or if the queue serves a wireless link or it is part of a larger scheduling hierarchy. Using units of time for the queue was taught by the CoDel algorithm [27] the year before. Because PIE was designed for hardware implementation, it did not measure the service time of the queue directly using the time-stamping approach of CoDel; instead it converted queue lengths to queuing delay using a regularly updated estimate of the link rate. Actually, as far back as 2002, Kwon and Fahmy [24] had advised that the queue should be measured in units of time. Also, in 2003, Sågfors *et al* had modified the Packet Discard Prevention Counter (PDPC+ [31]) algorithm by converting queue length to queuing delay to cope with the varying link rate of 3G wireless networks.

The second main enhancement introduced in PIE was to scale α & β dependent on the magnitude of p, which was based on the stability analysis in [28] (see equation (35) in Appendix B). The Bode margin plots in Figure 4 show the gain and phase margins for loss probabilities between 0.0001% and 100%, with a fixed maximum RTT and for different α and β parameters. With fixed α and β, the gain margin evolves diagonally as p varies with load. When the gain margin and the phase margin are negative, the control loop becomes unstable, resulting in oscillating queue size and TCP throughput. On the other hand, when the gain margin becomes too high the control loop responds too sluggishly to load changes. The auto-tuned plot shows the effect of introducing scaling steps in PIE to keep the margins above zero for lower p, while keeping the gain margin low for higher p. In the original PIE paper [28], no further scaling steps were applied for $p < 1\%$. In early versions of the IETF specification, scaling did not extend below 0.1%, but following criticism during the IETF review process that a table of values inherently limits the operating range of the algorithm [6], the table of values was extended down to 0.0001% as shown in Figure 5. Up to that point, the written justification for the scaling had been rather arbitrary. However, the extra data points confirmed suspicions that this scaling tracked the square-root law of TCP Reno, although it is not clear whether this was conscious design intent, or the outcome of empirical design iterations.

The third main enhancement introduced in PIE [28] was a burst allowance. This mechanism only allows bursts when the queue first becomes active after a period of no queuing, so it is inactive for bursts on top of a low level of queuing. The paper itself points out that the incremental evolution of drop probability p in the underlying PI algorithm already filters out anomalous bursts, but anyway adds an additional burst allowance that can be explicitly configured, if required for the deployment scenario.

The proportional-integral approach was generally considered sufficient for AQM control, with the proportional term ensuring loss increases more quickly if the queue is growing and less if it is shrinking, and the integral term returning the queue slowly enough to its target level. Nonetheless, proportional-derivative and proportional-integral-derivative controllers have been proposed. These are summarized in Adams [1], which provides a useful survey of a wide range of AQM schemes, with a particular focus on those derived from control-theoretic analysis.

As Adams [1] points out, all control theoretic approaches need to assume a model of how the traffic responds to control signals, and all known schemes (until now) assume that traffic consists predominantly of long-running TCP Reno flows.

Kühlewind *et al* [23] investigated the feasibility of using coupled AQM algorithms to deploy a scalable TCP congestion control such as Data Centre TCP [2] alongside existing TCP Reno traffic. It documented the throughput deviations between linearly coupled AQMs empirically, whereas we have derived the square relationship used in the present work analytically.

An IETF specification of a dual-queue coupled AQM has been drafted [13], to enable coexistence of 'Scalable' (e.g. DCTCP) and 'Classic' (e.g. TCP Reno) congestion controls. It requires the drop (or marking) probability of the AQM for 'Classic' traffic to be the square of that

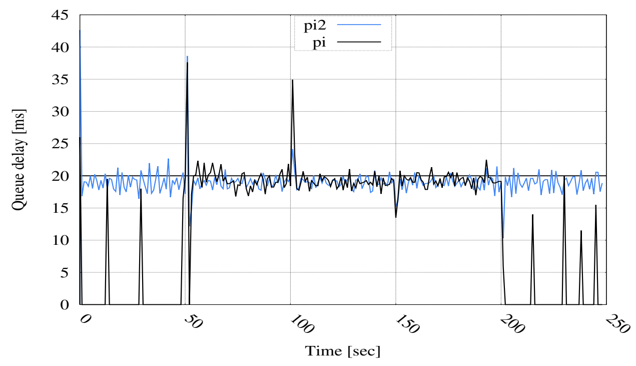

Figure 6: Performance comparison under varying traffic intensity: 10:30:50:30:10 TCP flows over durations 50:50:50:50:50 s, link capacity = 100 Mbps, RTT = 10 ms, α_{PI}=0.125, β_{PI}=1.25, α_{PI2}=0.3125, β_{PI2}=3.125, T=32ms.

for scalable traffic. It is written sufficiently generically that it covers the PI^2 approach, but the example AQM it gives is based on a RED-like AQM called Curvy RED.

The present work focuses on PI^2 in a single queue to fully evaluate it relative to existing single queue solutions before taking the next step of proposing its use in a dual-queue.

4. PI IMPROVED WITH A SQUARE

Restructuring PIE.

When the probability is small PIE tunes α and β to become smaller as well, in order to compensate for the higher sensitivity of the signal p. In Figure 6 the 'pi' curve shows what happens if the α and β parameters are not auto-tuned. For the lower load when there are 10 flows (between 0-50s and 200-250s) any onset of congestion is immediately suppressed very aggressively (p becomes too high, because β is too high), resulting in underutilization and an oscillating queue. Instead of directly PI-controlling the applied packet congestion notification probability (p), we propose to do the PI-control process on a pseudo-probability (p') in linear space and afterwards to adapt the controlled value to its non-linear form. In Figure 6 the 'pi2' curve shows what happens if constant (non-auto-tuned) α and β parameters are used, before the pseudo-probability is converted to its square and applied as a drop probability. The square ensures that the pseudo drop probability is adjusted to the right sensitivity level of the signal ($\sqrt{p}$) and it makes room for higher α and β, as we shall see.

The load that an AQM has to push back against is proportional to the number of TCP-like sources generating it. The total steady-state rate of arriving bits is not a measure of load, because all the TCP sources fit their total bit rate into the link capacity. Rather, the excess load is the rate at which each source adds packets to exceed the link capacity. For instance, TCP Reno adds one segment per RTT, so when the number of sources doubles the number of segments added per RTT doubles. TCP eventually ensures the rate of each flow halves, so load is inversely proportional to steady-state flow rate. In fact, for ACK-clocked sources like TCP, the load is proportional to $1/W$, where W is the steady-state window of each flow.

Therefore, for TCP Reno or CReno the load is proportional to $\sqrt{p}$ (from their steady-state window equations (5) and (7) in Appendix A). Therefore, we will control $p' = \sqrt{p}$. To transform the control output p' to the non-linear form needed for applying congestion notification we will simply square it ($p = (p')^2$).

Note that, even though p is smaller than p', for the same TCP load the PI^2 controller will drive the steady-state value of p' sufficiently high so that its value when squared (p) will be the same as that from PIE.

We shall now show that the heuristic table of scaling factors in PIE was actually attempting to achieve the same outcome as PI^2. We will abbreviate the main control terms in PIE (see (4) in section 2) as

$$\pi(\tau) \equiv \alpha\big(\tau(t) - \tau_0\big) + \beta\big(\tau(t) - \tau(t-T)\big)$$

Then our proposed squaring approach can be written:

$$p \leftarrow \big(p' + K\pi(\tau)\big)^2,$$

where K is a constant used later. Expanding:

$$\leftarrow (p')^2 + 2Kp'\pi(\tau) + K^2\pi^2(\tau).$$

Assuming $K\pi(\tau) \ll p'$, this approximates[1] to:

$$\leftarrow p + 2Kp'\pi(\tau).$$

As Figure 5 illustrates, PIE's stepped scaling factor 'tune' broadly fits the equation $\sqrt{2p}$. So we can say $2K_{\text{PIE}}p' \approx \sqrt{2p}$ or $K_{\text{PIE}} \approx 1/\sqrt{2}$. Thus the stepped way that PIE derives drop probability p is indeed broadly equivalent to incrementing p' using the core PI function then squaring the result.[2] Using the stability analysis in Appendix B we shall now see that we can make PI^2 more responsive than PIE without risking instability, because K_{PI2} can be greater than K_{PIE}.

Responsiveness without Instability.

In Misra *et al* [26] and Hollot *et al* [19] a fluid model is used to describe the TCP Reno throughput, the queuing process and the PI AQM behaviour, and to perform its stability analysis. Appendix B describes this analysis for TCP Reno over a PI^2 AQM. Based on the derived loop transfer functions (36) and (37), the Bode gain and phase margins are plotted in Figure 7 (the lines labelled 'reno pie' are the PIE auto-tuned margins, and 'reno pi2' the Reno on PI^2 margins).[3]

[1]In the rare cases when this inequality is not valid, it merely means that PI^2 will calculate a higher Δp than PIE would; neither is necessarily more 'correct'.
[2]PIE scales by the old value of p'; PI^2 scales by the new.
[3]The octave scripts that generated these plots and those in Figure 4 are available as auxiliary material with this paper in the ACM Digital Library.

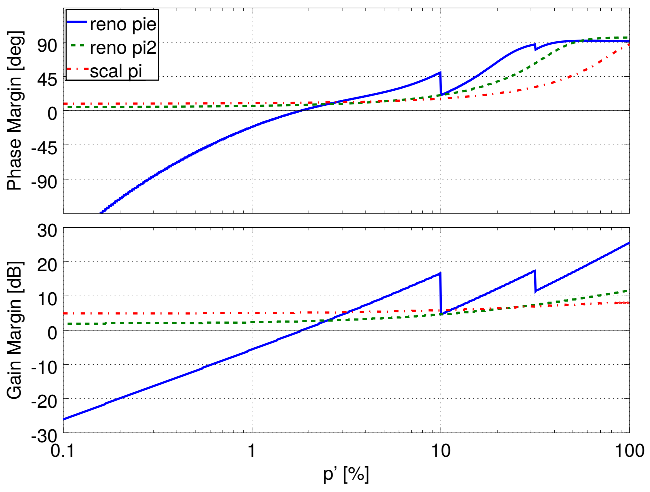

Figure 7: Bode plot margins for R=100ms, α_{PIE}=0.125*tune, β_{PIE}=1.25*tune, α_{PI2}=0.3125, β_{PI2}=3.125, α_{PI}=0.625, β_{PI}=6.25, T=32ms

Applying the squaring outside the PI controller flattens out the otherwise diagonal gain margin and makes it easy to choose values for the gain factors α and β that only depend on the maximum RTT to be supported. Only at high loads, when p' is higher than 60% ($p > 36\%$) is the gain margin of PI^2 slightly above 10dB. Because the gain margin of PI^2 is flatter, it can be made more responsive than PIE by increasing the gain factors by $\times 2.5$ without the gain margin dipping below zero anywhere over the full load range, which could otherwise lead to instability. This makes the gain of PI^2 roughly 3.5 times (or 5.5 dB) greater than that of PIE, because $K_{\text{PI2}}/K_{\text{PIE}} \approx 2.5\sqrt{2} \approx 3.5$.

PI^2 Design.

In Figure 8 the PI^2 AQM is shown. Compared to PIE, the scaling block is removed and the drop/mark decision block is modified to apply the squared drop probability. The squaring can be implemented either by multiplying p' by itself, or by comparing it with the maximum of 2 random variables during the drop decision. The first is easy to perform in a software implementation, perhaps as an addition to existing PIE hardware. The latter might be preferred for a hardware implementation. As the resolution of p' is half that of its square, the number of random bits used for each of the two random variables only needs to be half that of a PIE decision function.

Coexistence between Congestion Controls.

Additionally the above analysis suggests that control of 'Scalable' controls (see Section 2) such as DCTCP might also exhibit the same stability properties. Equation 11 in Appendix A shows that the window of DCTCP with probabilistic marking is inversely proportional to the drop or marking probability. So it should be possible to apply p' directly to DCTCP packets without the squaring during the drop/mark decision.

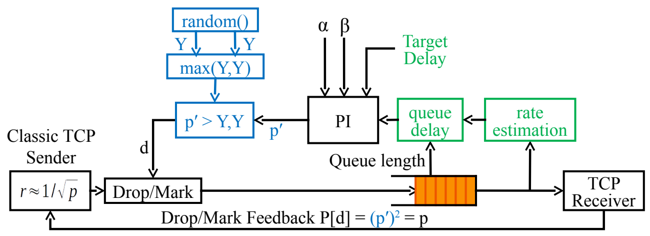

Figure 8: PI^2 for 'Classic' Traffic (PIE changes in blue)

This is confirmed analytically in Appendix B, where stability is analysed for a congestion control that reduces its window by half a packet per mark, which is a good approximation[4] for what DCTCP effectively does when a probabilistic AQM is used. Figure 7 shows its gain margins (lines labelled 'scal pi'). The plots are very similar to the Reno on PI^2 results but there was enough margin to double the α and β parameters (lines labelled 'reno pi2') relative to the Classic α and β parameters.

We have seen that PI^2 can be applied to Classic traffic and PI without the square can be applied to Scalable traffic. When both types of traffic coexist, it should also be possible to apply whichever of the two controls is appropriate, as long as each type of traffic can be distinguished. The drop/mark probability relation between DCTCP and CReno flows for an equal steady state throughput is derived in (14) (see Appendix A): $p_c = (p_s/k)^2$, with p_c the drop or mark probability for CReno flows, p_s the mark probability for Scalable (e.g. DCTCP) flows and k the coupling factor. In (14) a value of 1.19 is derived for k but it is set to 2 in the rest of this paper, having been validated empirically in the following section. Note that $k = 2$ is also the optimal ratio between the Scalable and Classic gain factors for optimal stability. As the gain factors determine the final probability, p_s will also be double $\sqrt{p_c}$, matching k.

5. PI^2 AQM IMPLEMENTATION

To verify the theoretical predictions, we have modified the PIE Linux AQM to support the PI^2 control method for classic congestion controls (Cubic, Reno, DCCP, SCTP, QUIC, ...), to support control of scalable congestion controls (DCTCP, Relentless, Scalable,...) and to support coexistence between the classic and scalable congestion controls. The Linux qdisc source code used for these experiments is released as open source and can be retrieved at [11]. Figure 9 shows the detailed implementation diagram.

To implement the PI^2 control method, we removed the code in PIE that scales the gain factors dependent on the current probability and instead added code to compare the resulting probability with two random variables.

[4]DCTCP additionally smooths (and consequently delays) the response from the network over a fixed number of RTTs which results in a more sluggish response for bigger RTTs.

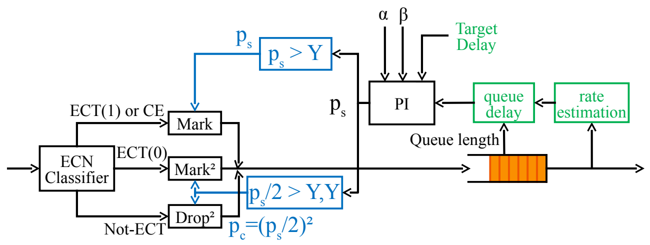

Figure 9: Coupled PI^2 and PI AQMs for Classic and Scalable Traffic in a Single Queue

To support Scalable congestion controls, we also made the squared drop decision conditional on a congestion control family identifier. We used the ECN bits in the IP header to distinguish packets from Scalable and Classic congestion controls.

The more frequent congestion signal induced by a Scalable control makes using drop for congestion signalling intolerable. In contrast, explicit congestion notification provides a congestion signal without also impairing packet delivery. Currently, DCTCP enables the ECN capability by setting the ECT(0) codepoint in the IP header and it expects any AQM in the network to mark congestion using a queue threshold that is both shallow and immediate (not delayed by smoothing).

We modified DCTCP to set the ECT(1) codepoint, which the IETF has recently agreed in principle to make available with a view to using it as the identifier for Scalable traffic [4, 14]. This would allow Classic congestion controls to continue to be able to support 'Classic' ECN by using the ECT(0) codepoint. It is proposed that both Classic and Scalable traffic would have to share the same Congestion Experienced (CE) codepoint to indicate a congestion marking. Then any congestion marking on ECT(0) packets would continue to have the same meaning as a drop. The pros and cons of using different identifiers are outside the scope of this paper, but they are discussed in the above-cited draft.

In the network, all packets use the same FIFO queue, but we classify packets based on their ECN codepoint in order to apply the appropriate drop/mark decision. We emphasize again that this single-queue approach is not intended or recommended for deployment; it is simply an interim step in the research process to avoid changing too many factors at once. To support coexistence we ensure rough throughput equality (a.k.a. TCP-fairness) by applying the $k = 2$ factor to Classic packets before making the drop/mark decision based on the squared probability.

Fewer Heuristics.

For our PI^2 implementation, as well as removing all the scaling heuristics, for PI^2 we additionally disabled the following heuristics that had all been added to the Linux implementation of PIE:

- Burst allowance: In PIE 100ms after the queue was last empty no drop or marking is applied. We disabled this (as also allowed in the PIE specification) so as not to influence DCTCP fairness, which is better when probabilities are constantly coupled (avoiding on-off behaviour).
- In PIE, if the probability is below 20% and the queue delay below half the target, no dropping or marking is applied. This rule was also disabled to avoid on-off behaviour. If this rule were not disabled, the threshold of 20% would have had to be corrected to the square root (around 45%).
- In PIE, if the probability is above 10%, ECN capable packets are dropped instead of marked. This is one possible overload strategy to prevent ECN traffic from overfilling the queue while starving drop based traffic. We disabled this rule and instead placed a maximum limit of 25% on the Classic mark/drop probability and the equivalent limit (100%) on Scalable marking probability. As a result the queue will be allowed to grow over the target if it cannot be controlled with this maximum drop probability. Then, if needed, tail-drop will control non-responsive traffic, whether ECN-capable or not.
- In PIE, if the probability is higher than 10% Δp is limited to 2%. We disabled this for now. Further comparative evaluation is needed to investigate the validity of this rule in the square root domain.
- In PIE, if the queue delay is greater than 250 ms, Δp is set to 2%. We also disabled this for now, again re-evaluation in the square root domain is needed.

So far, the basis on which most of these heuristics were chosen is undocumented and no-one has published an exhaustive assessment of the precise impact on performance of each. Therefore, given we had no sound basis to reconfigure each one to interwork with the restructured PI^2 mechanism, we chose to disable them.

The results for PIE shown in this paper are the ones for the full Linux PIE implementation, with all its heuristics except we reworked the rule that disables ECN marking above 10%, as described above. This avoided a discontinuity in the results for the throughput ratio between Classic and Scalable flows.

We produced an implementation of PIE without the above extra heuristics (called bare-PIE) and repeated all the experiments presented in this paper. We saw no difference in any experiment between bare-PIE and the full PIE. This gives confidence that the results of the PI^2 experiments are only due to the PI^2 improvements, and not due to disabling any of the above heuristics.

6. EVALUATION

Our evaluations of PI^2 consisted of two main sets of experiments: comparing the performance of PI^2 with PIE, and verifying our claims that PI^2 will enable coexistence of Scalable and Classic flows.

Our testbed consists of an AQM, 2 client and 2 server machines, all running Linux (Ubuntu 14.04 LTS with kernel 3.18.9), which contained the implementation of the TCP variants and AQMs.

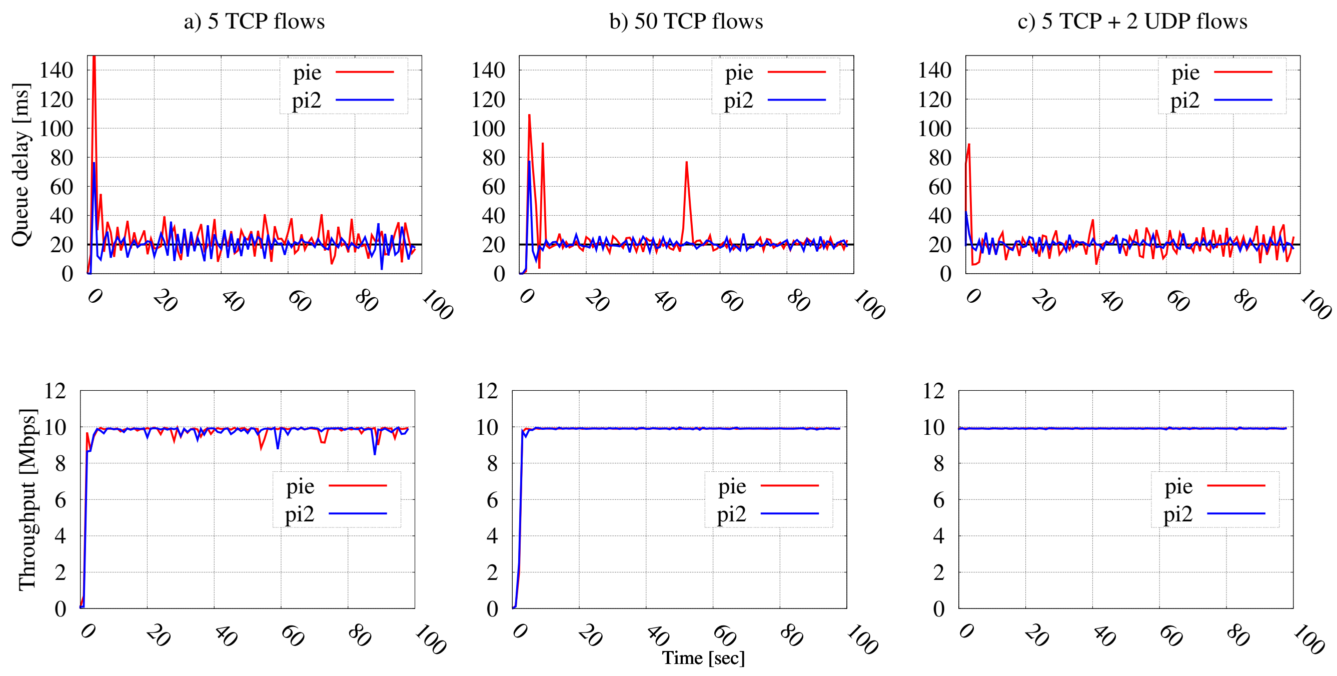

Figure 11: Queuing latency and throughput under various traffic loads: link capacity = 10 Mbps, RTT = 100 ms.

The experiments used unmodified DCTCP and Cubic implementations in their default configurations. For ECN-Cubic, we additionally enabled TCP ECN negotiation on the relevant client and server. The AQM configurations used the options as described in Table 1, unless otherwise stated.

A more detailed overview of how these machines are connected is presented in Figure 10. We used each client-server pair (Client A - Server A, Client B - Server B) for emulating TCP flows of the same congestion control (and UDP flows in addition, if necessary), so that the competition between different congestion control flows could be evaluated. For the experiments where only one congestion control was evaluated, only a single client-server pair was used.

Responsiveness and Stability.

For the first series of stability tests, we repeated the main experiments presented by Pan et al. [28]. We reproduced the PIE results and evaluated PI^2's performance compared to those of PIE. For reference, we will use the same test titles as those used by Pan et al. in their paper (highlighted with italics).

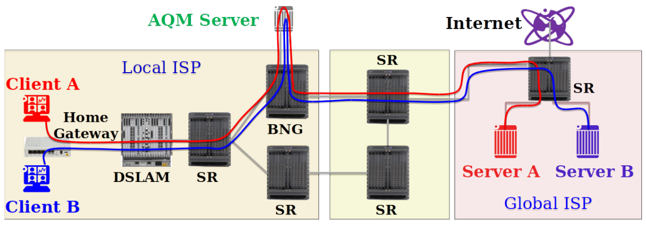

Figure 10: Testbed topology

All	Buffer: 40000 pkt (2.4 s @200 Mb/s), ECN
PI/PIE+Cubic/Reno	Target delay: 20 ms, Burst: 100 ms, α: 2/16, β: 20/16
PI/PI^2+DCTCP	Target delay: 20 ms, α: 10/16, β: 100/16

Table 1: Default parameters for the different AQMs.

We used TCP Reno, on a topology as shown in Figure 10. To reproduce the same environment, the bottleneck link capacity is 10 Mbps (except Figure 12 where we simulate varying link capacity) and the RTT is 100 ms if not specified otherwise. The sampling interval applied in the graphs is 1 s.

Figure 11 shows the queuing delay (upper row), and total throughput for all flows (lower row), where we use the following traffic mixes:

a) *Light TCP traffic*: 5 TCP flows.
b) *Heavy TCP traffic*: 50 TCP flows.
c) *Mixture of TCP and UDP traffic*: 5 TCP flows and 2 UDP flows sending at 6Mbps each

The results show less queue overshoot on start-up for PI^2 and less (damped) oscillations afterwards. This is because the flatter gain margin of PI^2 allowed us to set the responsiveness (gain) 3.5 times higher without risking instability (see section 4).

Figure 12 shows the queuing delay for the test with *Varying Link Capacity* as detailed in the caption. The PI^2 controller again seems to reduce overshoot and also when the link capacity is reduced the queue reduces faster and with less oscillation. The PIE AQM also allows overshoot when flow rates increase to fill the increased capacity at 100 s, while PI^2 shows no visible overshoot. Sampling at 100 ms intervals shows a peak queuing delay at 50 s of 510 ms for PIE and 250 ms for

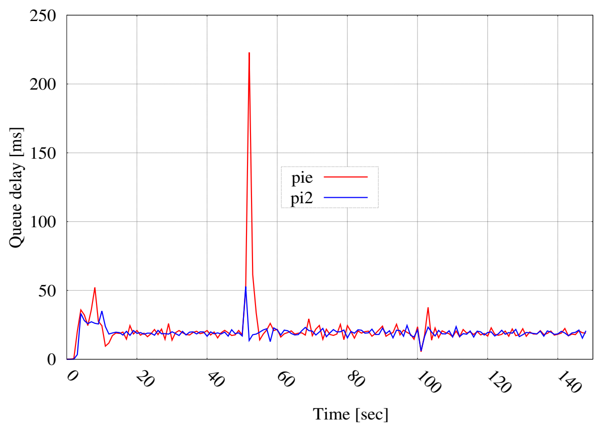

Figure 12: Performance comparison under varying link capacity: 100:20:100 Mb/s over durations 50:50:50 s

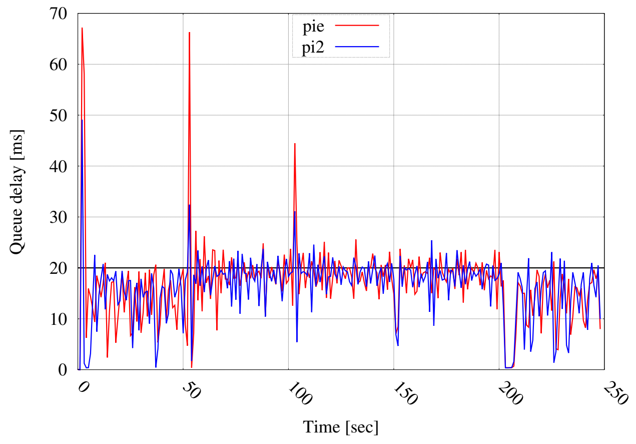

Figure 13: Performance comparison under varying traffic intensity: 10:30:50:30:10 TCP flows over durations 50:50:50:50:50 s, link capacity: 10 Mbps, RTT: 100 ms.

PI^2. PIE has 2 more oscillation peaks above 100 ms immediately afterwards, while PI^2 none at all.

Figure 13 demonstrates queuing delay for the *Varying Traffic Intensity* test as detailed in the caption. Again PI^2 reduces overshoot during load changes and also reduces upward fluctuations during non-transient periods.

Figure 14 shows the CDF of *Queuing Delay Comparison* between PI^2 and PIE, with target delay of 5 ms (upper row) and 20 ms (lower row). For these experiment, we use 20 TCP flows during the first part (a) and 5 TCP + 2 UDP flows during the second part (b). In all situations PI^2 has a similar queue size to PIE.

As we have seen, the improvement in responsiveness and stability of PI^2 is measurable but slight. Our ultimate deployment goal is a dualQ structure [13], in order to significantly reduce latency in the queue for Scalable traffic. The improvement in short flow completion time is evaluated in [12]. Using the single queue arrangement of the present paper, we experimented with mixed flow sizes over an extensive range of link rates and RTTs and, as expected, mixed short flow completion times with PIE, bare PIE and PI^2 under both heavy and light Web-like workloads were essentially the same.

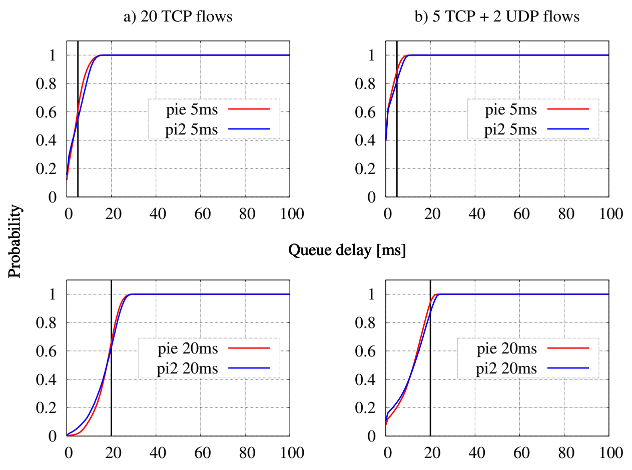

Figure 14: Queuing delay comparison under different traffic loads using 5 ms and 20 ms target delays: link capacity = 10 Mbps, RTT = 100 ms.

Coexistence.

The second series of tests was focused on coexistence between Classic and Scalable congestion controls with different traffic mixes and network conditions, exploring different combinations of link capacity and RTT.

We performed experiments with mixed Cubic and DCTCP flows and measured the ratio of their individual flow throughputs (Figure 15), and the average and 99th percentile of queue delay (Figure 16). We also monitored mark or drop probability (Figure 17) and link utilization (Figure 18) for each experiment.

In a first set of experiments, we used a single long-running flow for each congestion control. Figure 15 demonstrates throughput 'fairness'. As well as the Cubic/DCTCP throughput ratio (blue), we also included Cubic/ECN-Cubic (black) as a control to show the effect of the different congestion control algorithm separately from the addition of ECN. In both cases the ratio is non-ECN-capable to ECN-capable flow rate. As an illustration of the coexistence problem, the Cubic/DCTCP ratio with PIE shows, as expected, that DCTCP effectively starves Cubic, behaving about ten times more aggressively. In contrast, applying PI^2 to Cubic and PI to DCTCP (labelled PI2) works remarkably well over the whole range of link rates and RTTs, keeping the Cubic/DCTCP rate balance close to 1 in all cases, nearly exactly counterbalancing DCTCP's significantly greater aggression. In the control case of Cubic/ECN-Cubic, as expected, we see similar results for both PI^2 and PIE.[5]

As we have emphasized, the ultimate aim is to exploit the low queuing delay of scalable controls using a dualQ.

[5]This experiment uncovered a possible bug in Linux that currently limits the bandwidth-delay product (BDP) to 1 MB. Despite considerable effort, the cause is yet to be ascertained. Unfortunately this caused the anomalous results at the high RTT end of the higher link rates, not only for rate balance, but also for queuing delay, drop/mark probability and utilization.

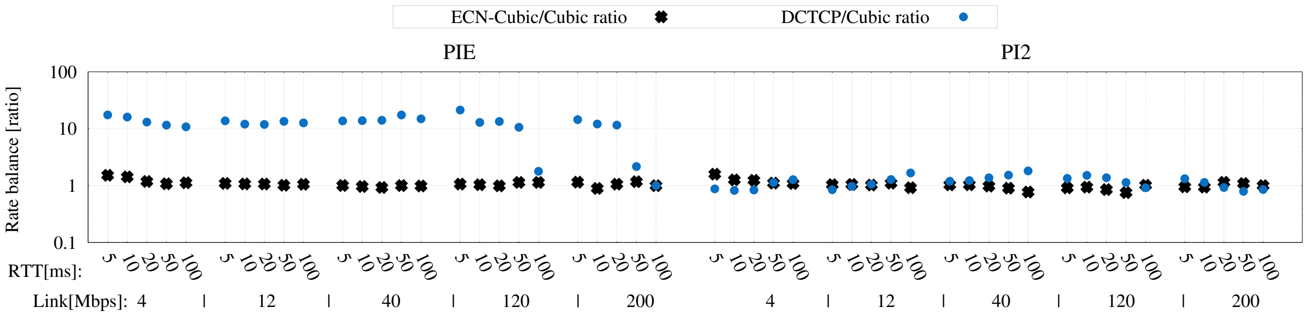

Figure 15: Throughput balance. One flow for each congestion control.[5]

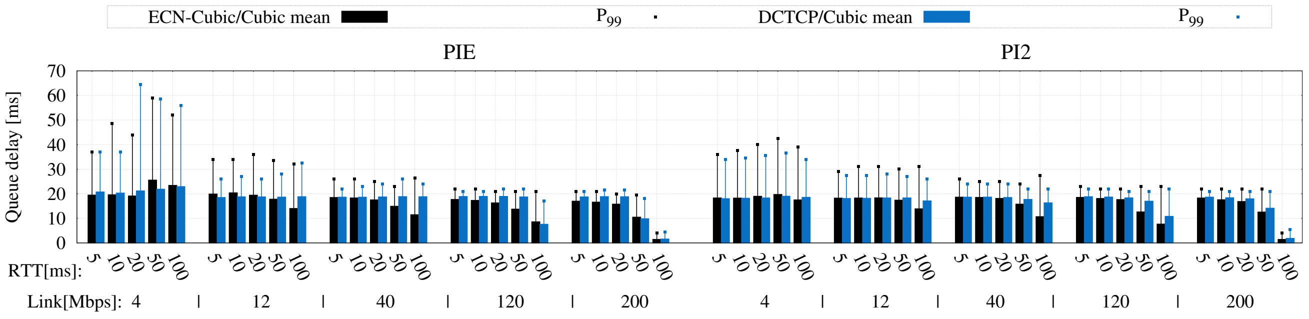

Figure 16: Queuing delay. One flow for each congestion control.[5]

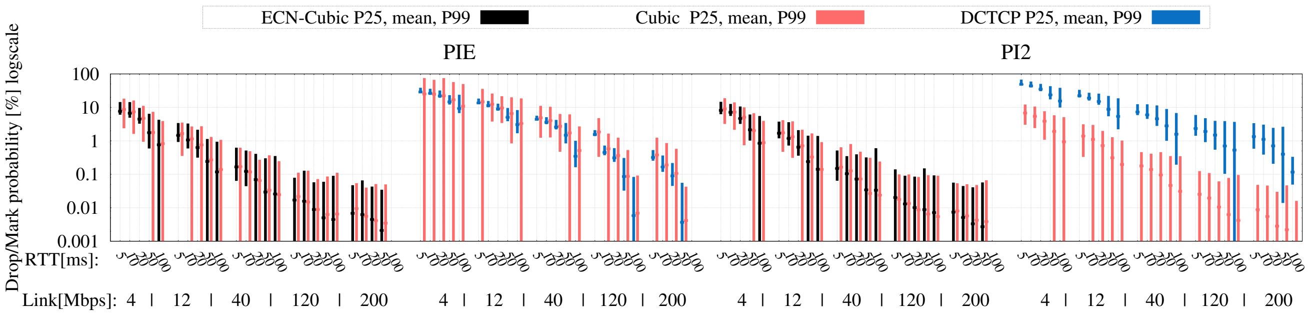

Figure 17: Marking/dropping probability. One flow for each congestion control.[5]

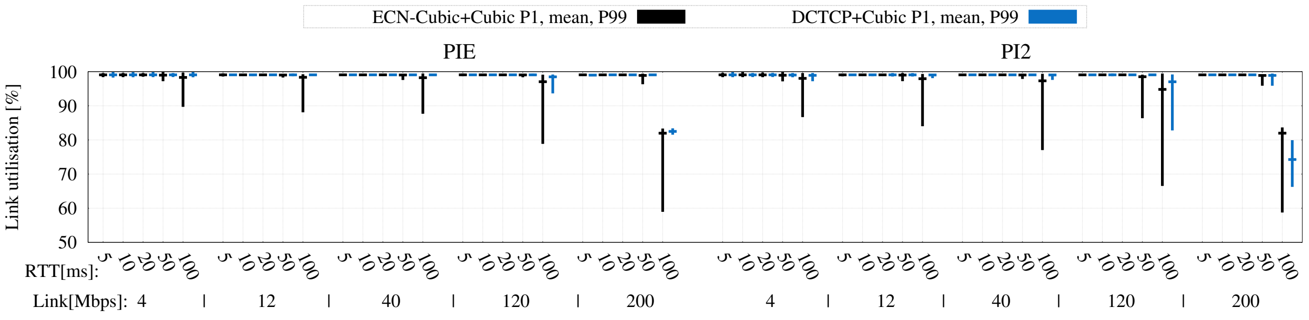

Figure 18: Link utilization. One flow for each congestion control.[5]

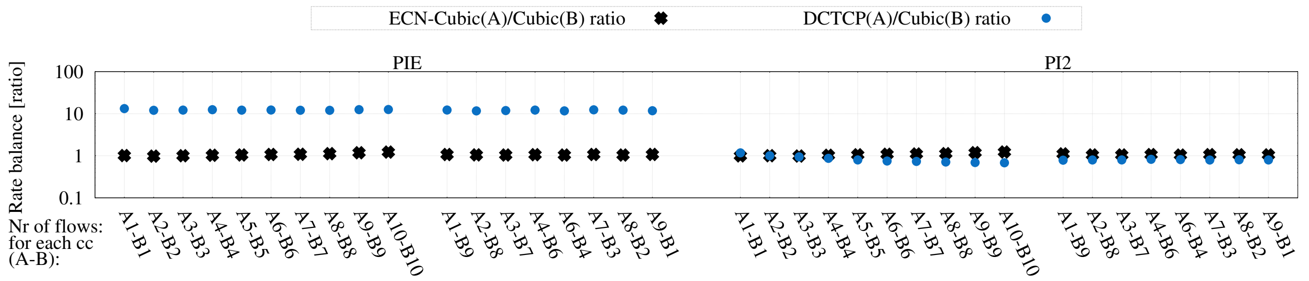

Figure 19: Throughput balance for different combinations of flows, link: 40 Mbps, RTT: 10 ms

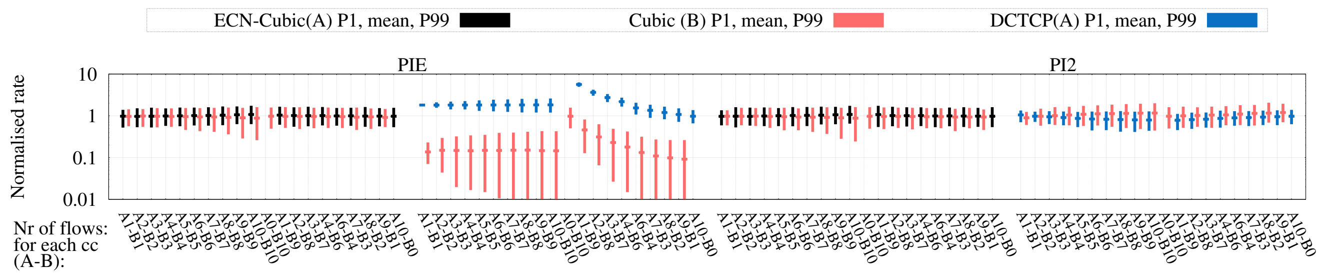

Figure 20: Throughput stability shown as normalized rate per flow (rate per flow divided by ‘fair’ rate) for different combinations of flows, link: 40 Mbps, RTT: 10 ms

With the single-queue approach studied in this paper, the queuing delay experienced by each type of concurrent flow had to be the same. In Figure 16 we compare the average and 99th %-ile (P_{99}) of queue delay for each packet between the two scenarios: a Cubic flow with an ECN-Cubic or a DCTCP flow. As expected, there is little difference between the scenarios. Nonetheless, the results indicate that PI^2 appears to be no worse than PIE at keeping the queue delay at the target of 20ms. PI^2 even seems to outperform PIE at smaller link rates, which can be seen at the link rate of 4Mbps, where P_{99} is larger for PIE.

To verify whether the throughput ratio is influenced by the number of concurrent flows, we tested it with a range of combinations of flow numbers. Figure 19 shows the per-flow throughput ratio for the combinations of flows displayed on the x-axis, where each combination is represented by two numbers. The first is the number of Cubic flows, while the second is the number of ECN-Cubic or DCTCP flows respectively. The results were similar for different link capacities and RTTs, so we show results for a link capacity of 40 Mbps and 10 ms RTT as an example. Figure 20 visualizes the same experiment, but includes the mean and %-iles by showing normalized rates. The results are very similar to those in Figure 15 showing that PI^2 can maintain equal flow rates irrespective of the number of concurrent flows.

7. CONCLUSIONS

The main conclusions of this work are twofold. Firstly, despite PI^2 being simpler than PIE, it achieves no worse, and in some cases better performance, particularly superior responsiveness during dynamics without risking instability. This has been demonstrated both analytically and through a number of experiments. In other words, the heuristic scaling steps introduced by PIE can be replaced by squaring the output instead, which is less computationally expensive and improves stability in some cases.

Secondly, in contrast to PIE, a combination of PI and PI^2 can support coexistence of Scalable and Classic congestion controls on the public Internet by counterbalancing the more aggressive congestion response of Scalable controls with more aggressive congestion marking. It has been emphasized throughout that the arrangement of both PI and PI^2 in the same queue evaluated in this paper is only a step in the research process, not a recommended deployment. The recommended deployment applies each AQM to separate queues [13] so that Scalable traffic in the PI-controlled queue can maintain extremely low latency while isolated from but coupled to the queue built by Classic traffic controlled by PI^2. A complementary paper explains and evaluates this ‘DualQ Coupled’ approach [12].

Acknowledgement

The authors’ contributions were part-funded by the European Community under its Seventh Framework Programme through the Reducing Internet Transport Latency (RITE) project (ICT-317700). The views expressed here are solely those of the authors.

APPENDIX

A. EQUAL STEADY STATE RATE

Classic TCPs have a throughput equation that is proportional to the square root of the signal probability. For our purposes, it is sufficient to ignore dynamic aspects, and apply the simplified models in (5) and (6) for TCP Reno and Cubic window as in [25] and [16]:

$$W_{reno} = \frac{1.22}{p^{1/2}} \qquad (5) \qquad\qquad W_{cubic} = \frac{1.17R^{3/4}}{p^{3/4}} \qquad (6)$$

where p is drop probabillity and R is the RTT.

As we said, the Cubic implementation in Linux falls back to TCP Reno with a different decrease factor ($B = 0.7$ in Linux), thus the CReno steady state window will deviate from (5) with a slightly higher constant (7).

The implicit switch-over RTT can be derived from (6) and (7). Pure Cubic behaviour (as defined in (6)) will become active when (8) is false.

$$W_{creno} = \frac{1.68}{p^{1/2}} \qquad (7) \qquad\qquad W * R^{3/2} < 3.5 \qquad (8)$$

This shows that the switch-over point does not depend on a single RTT or BDP (bandwidth delay product) value, but a more complex combination of RTT and window (W).

In this paper we focus on fairness between Scalable flows and Cubic flows in their Reno mode (CReno). For the Scalable TCP we use DCTCP with probabilistic (not on-off) marking applied by the AQM. We have derived the steady-state window equation for DCTCP in this case, assuming an idealized uniform deterministic marker, which marks every $1/p$ packets. A DCTCP congestion controller has an incremental window increase per RTT $a = 1$ and a multiplicative decrease factor $b = p/2$ (with p being estimated). So, every RTT, W is increased by $W \leftarrow W + 1$, meaning that under steady state, this must be compensated every RTT by (9). This decrease is steered by ECN marks, as defined in (10).

$$W \leftarrow \left(1 - \frac{1}{W}\right) W \quad (9) \qquad W \leftarrow \left(1 - \frac{p}{2}\right) W \quad (10)$$

From (9) and (10), we see that to preserve this balance, window equation (11) must be true.

$$W_{dc} = \frac{2}{p} \quad (11) \qquad W_{dcth} = \frac{2}{p^2} \quad (12)$$

Note that (12) derived in the DCTCP paper [2] has a different exponent of p compared to (11). The reason is that (12) is defined for a step threshold, which causes an on-off pattern of RTT length marking trains. In contrast, when marking for DCTCP is steered by a PI controller, its random process with a fractional probability will cause an evenly distributed marking pattern, so (11) will be applicable. This explains the same phenomenon found empirically in Irteza *et al* [22], when comparing a step threshold with a RED ramp.

A Scalable congestion controller such as DCTCP achieves low throughput variations by driving the network to give it a more responsive signal with a higher resolution. Therefore, a solution must be found to reduce the congestion signal intensity for Classic congestion controllers (TCP Cubic and Reno), balanced with that for DCTCP.

Knowing the relation between network congestion signal (mark or drop) probability and window, we can adjust feedback from the network to each type of congestion control. For TCP CReno and DCTCP, we substitute (7) and (11) in $W_{creno} = W_{dc}$:

$$\frac{1.68}{p_{creno}^{1/2}} = \frac{2}{p_{dc}} \quad (13) \qquad p_{creno} = \left(\frac{p_{dc}}{1.19}\right)^2 \quad (14)$$

Therefore, if the RTTs are equal, we can arrange the rates to be equal using the simple relation between the probabilities, defined in (14).

Probabilistic mark/drop is typically implemented by comparing the probability p with a pseudo-randomly generated value Y per packet. A signal is applied for a packet when $Y < p$. The advantage of using relation (14) is that p^2 can easily be acquired by comparing p with 2 pseudo-random generated values and signalling only if both random values are smaller than p: $max(Y_1, Y_2) < p$.

The phrase "Think once to mark, think twice to drop" is a useful *aide-mémoire* for this approach, because Scalable controls always uses ECN marking (the marking level is often too high to use drop), while Classic controls typically use drop.

B. FLUID MODEL

In Pan *et al* [29] a PIE controller was designed for TCP Reno. TCP Reno has a throughput that is proportional to $1/\sqrt{p}$. In this analytical section we define first a model for a system that has an adaptor on the output of the controller that squares the signal, so the controller generates a signal $p' = \sqrt{p}$ which is finally applied to the packets as a marking or dropping signal $p = (p')^2$.

Secondly we define a system where the TCP is defined as a so-called scalable congestion control (section 2) with a throughput that is proportional to $1/p'$, so no squaring is needed at the end. We will consistently use p' to indicate a scalable probability and p to indicate a classic probability. One can be derived from the other with the above equation.

We start from the model of the evolution of the window of TCP Reno and the dynamics of the queue from Misra *et al* [26] and [19]:

$$\frac{dW(t)}{dt} = \frac{1}{R(t)} - 0.5\frac{W(t)W(t-R(t))}{R(t-R(t))}p(t-R(t)), \quad (15)$$

$$\frac{dq(t)}{dt} = \frac{W(t)}{R(t)}N(t) - C(t), \quad (16)$$

where $W(t)$ is the window size, $q(t)$ the queue size, R(t) the harmonic mean of the round trip time of the different flows, $N(t)$ is the number of flows, and C(t) is the link capacity, which might also vary over time, but is here assumed independent from all other variables.

The Cubic implementation in Linux provides a fall-back to TCP Reno when RTT or rate is small (CReno). For CReno mode the multiplicative reduction factor is 0.7 instead of 0.5, which results in an equation that is just slightly different:

$$\frac{dW(t)}{dt} = \frac{1}{R(t)} - 0.7\frac{W(t)W(t-R(t))}{R(t-R(t))}p(t-R(t)). \quad (17)$$

For TCP Reno that is steered by a squared probability, the following equation is used to derive the transfer function:

$$\frac{dW(t)}{dt} = \frac{1}{R(t)} - 0.5\frac{W(t)W(t-R(t))}{R(t-R(t))}\left(p'(t-R(t))\right)^2. \quad (18)$$

Doing the similar linearization exercise for (18) as for (15) in [19], assuming for now $N(t) = N$, $C(t) = C$, $R(t) = R_0$ as constants, we get operating point equations (defined by $dW/dt = 0$, $dq/dt = 0$ and

$(W,q,p')=(W_0,q_0,p'_0)$):

$$W_0^2 p_0'^2 = 2, \quad W_0 = \frac{R_0 C}{N}, \quad R_0 = \frac{q_0}{C} + T_p, \tag{19}$$

where T_p is the base delay of a specific flow.

The partial derivations are the same as in Appendix I of [19] except:

$$\frac{\partial f}{\partial p'} = -\frac{W_0^2}{2R_0}2p'_0 = -\frac{\sqrt{2}C}{N},$$

where $f(W, W_R, q, p')$ is defined as the RHS of Equation 18, and $W_R = W(t-R)$. Note that

$$\frac{\partial f}{\partial W} = \frac{\partial f}{\partial W_R} = \frac{-W_0}{2R_0}p_0'^2 = \frac{-W_0}{2R_0}\frac{2}{W_0^2} = \frac{-1}{R_0 W_0} = \frac{-N}{R_0^2 C},$$

$$\frac{\partial f}{\partial q} = -\frac{1}{R_0^2 C} + \frac{W_0^2 p_0'^2}{2R_0^2 C} = -\frac{1}{R_0^2 C} + \frac{2}{2R_0^2 C} = 0,$$

both of which initially have different terms to the analysis in [19], but eventually resolve to the same result as [19]. As a result, the linearized equation for a Reno TCP with a squared p' and its Laplace transform will be:

$$\frac{d(\delta W(t))}{dt} = -\frac{N}{R_0^2 C}(\delta W(t) + \delta W_R) - \frac{\sqrt{2}C}{N}\delta p'(t-R_0); \tag{20}$$

$$sW(s) = -\frac{N}{R_0^2 C}W(s)(1+e^{-sR_0}) - \frac{\sqrt{2}C}{N}p'(s)e^{-sR_0}. \tag{21}$$

Also for a scalable TCP that reduces its current window by half a packet per mark, the following equation can be derived in a similar way:

$$\frac{dW(t)}{dt} = \frac{1}{R(t)} - 0.5\frac{W(t-R(t))}{R(t-R(t))}p'(t-R(t)). \tag{22}$$

The difference is that the current window size $W(t)$ is not present as it is not used to determine the window reduction. This makes the reduction term only dependent on the state at time $t - R(t)$.

Re-running the linearization exercise for (22) as in [19], we get operating point equations:

$$W_0 p'_0 = 2, \quad W_0 = \frac{R_0 C}{N}, \quad R_0 = \frac{q_0}{C} + T_p. \tag{23}$$

The partial derivations are the same as in [19] except for:

$$\frac{\partial f}{\partial W} = 0; \quad \frac{\partial f}{\partial p'} = -\frac{W_0}{2R_0} = -\frac{C}{2N}.$$

Note that

$$\frac{\partial f}{\partial W_R} = -\frac{1}{2R_0}p'_0 = -\frac{1}{2R_0}\frac{2}{W_0} = -\frac{1}{R_0 W_0} = -\frac{N}{R_0^2 C},$$

which initially had different terms to the analysis in [19], but eventually resolves to the same result as [19]. As a result, the linearized equations for a scalable TCP and its Laplace transform will be:

$$\frac{d(\delta W(t))}{dt} = -\frac{N}{R_0^2 C}\delta W(t-R_0) - \frac{C}{2N}\delta p'(t-R_0); \tag{24}$$

$$sW(s) = -e^{-sR_0}\big(\frac{N}{R_0^2 C}W(s) - \frac{C}{2N}p'(s)\big). \tag{25}$$

From [26] and [19] we repeat the linearized queue equation, its Laplace transform and Laplace transfer function:

$$\frac{d(\delta q(t))}{dt} = \frac{N}{R_0}\delta W(t) - \frac{1}{R_0}\delta q(t); \tag{26}$$

$$sq(s) = \frac{N}{R_0}W(s) - \frac{1}{R_0}q(s), \tag{27}$$

$$P_{queue}(s) = \frac{q(s)}{W(s)} = \frac{\frac{N}{R_0}}{s + \frac{1}{R_0}}. \tag{28}$$

The PI transfer function in the Laplace domain is:

$$C_{PI}(s) = \frac{\frac{\beta+\alpha/2}{C}s + \frac{\alpha}{TC}}{s}. \tag{29}$$

The combined PI and queue (=AQM) transfer function in terms of R_0 and W_0 is:

$$A(s) = \frac{(\beta+\frac{\alpha}{2})s + \frac{\alpha}{T}}{s}\frac{\frac{1}{W_0}}{s+\frac{1}{R_0}}, \tag{30}$$

$$A(s) = \frac{\kappa_A}{W_0}\frac{(s/z_A + 1)}{s(s/s_A+1)}, \tag{31}$$

with $\kappa_A = \alpha R_0/T$, $z_A = \alpha/(T(\beta+\alpha/2))$ and $s_A = 1/R_0$.

The transfer functions of the different TCP and marking combinations in terms of W_0, R_0 and p_0 or p'_0 are:

$$P_{reno_p}(s) = \frac{W(s)}{p(s)} = -\frac{W_0\kappa_R e^{-sR_0}}{s/s_R + (1+e^{-sR_0})/2}; \tag{32}$$

$$P_{reno_{p'2}}(s) = \frac{W(s)}{p'(s)} = -\frac{W_0\kappa_S e^{-sR_0}}{s/s_R + (1+e^{-sR_0})/2}; \tag{33}$$

$$P_{scal_{p'}}(s) = \frac{W(s)}{p'(s)} = -\frac{W_0\kappa_S e^{-sR_0}}{s/s_S + e^{-sR_0}}, \tag{34}$$

with $\kappa_S = 1/p'_0$, $s_S = p'_0/(2R_0)$, $\kappa_R = 1/(2p_0) = \kappa_S^2/2$ and $s_R = \sqrt{2}p'_0/R_0 = \sqrt{2p_0}/R_0 = \sqrt{8}s_S$.

The complete loop transfer functions are:

$$L_{reno_p}(s) = \frac{\kappa_R\kappa_A(s/z_A+1)e^{-sR_0}}{(s/s_R + (1+e^{-sR_0})/2)(s/s_A+1)s}; \tag{35}$$

$$L_{reno_{p'2}}(s) = \frac{\kappa_S\kappa_A(s/z_A+1)e^{-sR_0}}{(s/s_R + (1+e^{-sR_0})/2)(s/s_A+1)s}; \tag{36}$$

$$L_{scal_{p'}}(s) = \frac{\kappa_S\kappa_A(s/z_A+1)e^{-sR_0}}{(s/s_S + e^{-sR_0})(s/s_A+1)s}. \tag{37}$$

8. REFERENCES

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

[2] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data Center TCP (DCTCP). *Proc. ACM SIGCOMM'10, Computer Communication Review*, 40(4):63–74, Oct. 2010.

[3] M. Alizadeh, A. Javanmard, and B. Prabhakar. Analysis of DCTCP: Stability, Convergence, and Fairness. In *Proceedings of the ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '11, pages 73–84, New York, NY, USA, 2011. ACM.

[4] D. Black. Explicit Congestion Notification (ECN) Experimentation. Internet Draft draft-black-tsvwg-ecn-experimentation-00, Internet Engineering Task Force, Sept. 2016. (Work in Progress).

[5] O. Bondarenko, K. De Schepper, I.-J. Tsang, B. Briscoe, A. Petlund, and C. Griwodz. Ultra-Low Delay for All: Live Experience, Live Analysis. In *Proc. ACM Multimedia Systems; Demo Session*, pages 33:1–33:4. ACM, May 2016.

[6] B. Briscoe. Review: Proportional Integral controller Enhanced (PIE) Active Queue Management (AQM). Technical Report TR-TUB8-2015-001, BT, May 2015.

[7] B. Briscoe, A. Brunstrom, A. Petlund, D. Hayes, D. Ros, I.-J. Tsang, S. Gjessing, G. Fairhurst, C. Griwodz, and M. Welzl. Reducing Internet Latency: A Survey of Techniques and their Merits. *IEEE Communications Surveys & Tutorials*, 18(3):2149–2196, 2016.

[8] B. Briscoe (Ed.), K. De Schepper, and M. Bagnulo. Low Latency, Low Loss, Scalable Throughput (L4S) Internet Service: Problem Statement. Internet Draft draft-briscoe-tsvwg-aqm-tcpm-rmcat-l4s-problem-02, Internet Engineering Task Force, July 2016. (Work in Progress).

[9] CableLabs. Data-Over-Cable Service Interface Specifications DOCSIS® 3.1; MAC and Upper Layer Protocols Interface Specification. Specification CM-SP-MULPIv3.1-I01-131029, CableLabs, Oct. 2013.

[10] Y. Choi, J. A. Silvester, and H.-c. Kim. Analyzing and Modeling Workload Characteristics in a Multiservice IP Network. *Internet Computing, IEEE*, 15(2):35–42, March 2011.

[11] K. De Schepper and O. Bondarenko. Linux sch_pi2 qdisc source code available at https://github.com/olgabo/dualpi2, June 2016.

[12] K. De Schepper, O. Bondarenko, I.-J. Tsang, and B. Briscoe. 'Data Centre to the Home': Deployable Ultra-Low Queuing Delay for All. Sept. 2016. (Under Submission).

[13] K. De Schepper, B. Briscoe (Ed.), O. Bondarenko, and I.-J. Tsang. DualQ Coupled AQM for Low Latency, Low Loss and Scalable Throughput. Internet Draft draft-briscoe-aqm-dualq-coupled-01, Internet Engineering Task Force, Mar. 2016. (Work in Progress).

[14] K. De Schepper, B. Briscoe (Ed.), and I.-J. Tsang. Identifying Modified Explicit Congestion Notification (ECN) Semantics for Ultra-Low Queuing Delay. Internet Draft draft-briscoe-tsvwg-ecn-l4s-id-01, Internet Engineering Task Force, Mar. 2016. (Work in Progress).

[15] N. Finn (Ed.). IEEE Standard for Local and Metropolitan Area Networks—Virtual Bridged Local Area Networks - Amendment: 10: Congestion Notification. Standard 802.1Qau, IEEE, Apr. 2010.

[16] S. Ha, I. Rhee, and L. Xu. CUBIC: a new TCP-friendly high-speed TCP variant. *SIGOPS Operating Systems Review*, 42(5):64–74, July 2008.

[17] O. Hohlfeld, E. Pujol, F. Ciucu, A. Feldmann, and P. Barford. A QoE Perspective on Sizing Network Buffers. In *Proc. Internet Measurement Conf (IMC'14)*, pages 333–346. ACM, Nov. 2014.

[18] C. V. Hollot, V. Misra, D. Towsley, and W. Gong. Analysis and design of controllers for AQM routers supporting TCP flows. *IEEE Transactions on Automatic Control*, 47(6):945–959, Jun 2002.

[19] C. V. Hollot, V. Misra, D. F. Towsley, and W. Gong. A Control Theoretic Analysis of RED. In *Proc. INFOCOM 2001. 20th Annual Joint Conf. of the IEEE Computer and Communications Societies.*, volume 3, pages 1510—19, 2001.

[20] Y. Hong and O. W. W. Yang. Self-tuning TCP traffic controller using gain margin specification. *IET Communications*, 1(1):27–33, February 2007.

[21] Y. Hong, O. W. W. Yang, and C. Huang. Self-tuning PI TCP flow controller for AQM routers with interval gain and phase margin assignment. In *Global Telecommunications Conference (Globecom'04)*, volume 3, pages 1324–1328, 2004.

[22] S. Irteza, A. Ahmed, S. Farrukh, B. Memon, and I. Qazi. On the Coexistence of Transport Protocols in Data Centers. In *Proc. IEEE Int'l Conf. on Communications (ICC 2014)*, pages 3203–3208, June 2014.

[23] M. Kühlewind, D. P. Wagner, J. M. R. Espinosa, and B. Briscoe. Using Data Center TCP (DCTCP) in the Internet. In *Proc. Third IEEE Globecom Workshop on Telecommunications*

Standards: From Research to Standards, pages 583–588, Dec. 2014.

[24] M. Kwon and S. Fahmy. A Comparison of Load-based and Queue-based Active Queue Management Algorithms. In *Proc. Int'l Soc. for Optical Engineering (SPIE)*, volume 4866, pages 35–46, 2002.

[25] M. Mathis, J. Semke, J. Mahdavi, and T. Ott. The macroscopic behavior of the TCP Congestion Avoidance algorithm. *Computer Communication Review*, 27(3), July 1997.

[26] V. Misra, W.-B. Gong, and D. Towsley. Fluid-based Analysis of a Network of AQM Routers Supporting TCP Flows with an Application to RED. *SIGCOMM Computer Comms.. Review*, 30(4):151–160, Aug. 2000.

[27] K. Nichols and V. Jacobson. Controlling queue delay. *ACM Queue*, 10(5), May 2012.

[28] R. Pan et al. PIE: A lightweight control scheme to address the bufferbloat problem. In *Proc. IEEE Int'l Conf. on High Performance Switching and Routing (HPSR)*, pages 148–155, 2013.

[29] R. Pan, P. Natarajan, F. Baker, G. White, B. Ver Steeg, M. Prabhu, C. Piglione, and V. Subramanian. PIE: A Lightweight Control Scheme To Address the Bufferbloat Problem. Internet Draft draft-ietf-aqm-pie-10, Internet Engineering Task Force, Sept. 2016. (Work in progress).

[30] K. K. Ramakrishnan, S. Floyd, and D. Black. The Addition of Explicit Congestion Notification (ECN) to IP. Request for Comments RFC 3168, RFC Editor, Sept. 2001.

[31] M. Sågfors, R. Ludwig, M. Meyer, and J. Peisa. Buffer Management for Rate-Varying 3G Wireless Links Supporting TCP Traffic. In *Proc Vehicular Technology Conference*, Apr. 2003.

[32] G. White. Active Queue Management Algorithms for DOCSIS 3.0; A Simulation Study of CoDel, SFQ-CoDel and PIE in DOCSIS 3.0 Networks. Technical report, CableLabs, Apr. 2013.