QoE: As Easy As PIE

Rong Pan, Preethi Natarajan, Chiara Piglione, Mythili S. Prabhu, Alon Bernstein
Advanced Architecture & Research Group,
Cisco Systems Inc., San Jose, CA 95134, U.S.A.

***Abstract*—Bufferbloat is a phenomenon where excess buffers in the network, such as in cable modems or wireless APs, cause high latency and jitter. As more and more interactive applications (e.g. voice over IP and real time video conferencing) run in the Internet, high latency and jitter degrade application performance and impact users' Quality of Experience (QoE). There is a pressing need to design intelligent queue management schemes that can control latency and jitter; and hence provide desirable quality of service to users.**

We present here a lightweight design, PIE (Proportional Integral controller Enhanced), that can effectively control the average queueing latency to a reference value. The design does not require per-packet extra processing, so it incurs very small overhead and is simple to implement in both hardware and software. In addition, the design parameters are self-tuning, and hence PIE is robust and optimized for various network scenarios. We apply the algorithm in DOCSIS 3.0 environment. Simulation results show that PIE can ensure low latency and achieve high link utilization under various congestion situations.

***Index Terms*—bufferbloat, Active Queue Management (AQM), Quality of Service (QoS), Quality of Experience (QoE), Explicit Congestion Notification (ECN)**

I. Introduction

The explosion of smart phones, tablets and video traffic in the Internet brings about a unique set of challenges for congestion control. To avoid packet drops, many service providers or data center operators require vendors to put in as much buffer as possible. With rapid decrease in memory chip prices, these requests are easily accommodated to keep customers happy. However, the above solution of large buffers fails to take into account the nature of TCP, the dominant transport protocol running in the Internet. The TCP protocol continuously increases its sending rate and causes network buffers to fill up. TCP cuts its rate only when it receives a packet drop or mark that is interpreted as a congestion signal. However, drops and marks usually occur when network buffers are full or almost full. As a result, excess buffers, initially designed to avoid packet drops, would lead to highly elevated queueing latency and jitter. The phenomenon was detailed in 2009 [1] and the term, "bufferbloat" was introduced by Jim Gettys in late 2010 [2]. Recent studies of home access network also confirmed that modems often have large buffers and that DSL links often have large high latency [3], [4].

Active queue management (AQM) schemes, such as RED [5], BLUE [6], PI [7], AVQ [8], etc, have been around for well over a decade. By selectively dropping packets early to optimize traffic behaviors, AQM schemes could potentially solve the aforementioned problem. RFC 2309 [9] strongly recommends the adoption of AQM schemes in the network to improve the performance of the Internet. Although controlling delay is a key metric in assuring QoS, the DOCSIS specifications do not define the adoption of AQM and only recently added the option to set a small queue size. The rationale behind it was that DOCSIS does not need AQM because DOCSIS addresses head-of-line blocking with fine-grained queuing or scheduling, i.e. sorting traffic into individual queue and then scheduling the drain of the queues according to the priority of these queues. In this way, the delay of the high priority traffic is minimized since it does not incur the delay issues caused by head-of-line blocking.

There are a few issues with the above line of thinking. First, head-of-line blocking is not the only cause for delay. Even if there is no head-of-line blocking, a large queue can still cause a large delay. For example, voice traffic can be sorted to one queue and data traffic can be put into a separate queue so there is no blocking of voice traffic. Nonetheless the data queue can still experience excessive delay. Second, limiting the buffer size is not the solution to the bufferbloat problem because latency is still high under persistent congestion. In addition, small buffer does not have enough space to absorb short-term

bursts, which could lead to throughput loss. Third, even with today's technology, per flow queueing is still rather complicated to implement. Except for traffic class like voice, a lot of flows are queued into to the same queue. For example, a background e-mail download can still create head-of-line blocking for a web browsing session. Latency control for this data queue is still required in order to guarantee QoE. Due to aforementioned issues, we believe that AQM is crucial in controlling latency and solving the bufferbloat problem.

Although AQM has not been adopted in the DOCSIS specifications, RED, a well-known AQM scheme, has been adopted in a wide variety of network devices, such as switches and routers; and it has been implemented in both hardware and software. Unfortunately, due to the fact that RED needs careful tuning of its parameters for various network conditions, most network operators do not turn RED on. In addition, RED is designed to control the queue length which would affect delay implicitly. It does not control latency directly.

We recognize that the delay bloat caused by poorly managed big buffers is the core issue here. If latency can be controlled, bufferbloat, i.e., adding more buffers for bursts, is not a problem. More buffer space would allow larger bursts of packets to pass through as long as we control the average queueing delay to be small. Unfortunately, Internet today still lacks an effective design that can control buffer latency to improve QoE of latency-sensitive applications. In addition, it is a delicate balancing act to design a queue management scheme that not only allows short-term burst to smoothly pass, but also controls the average latency when long-term congestion persists.

Recently, a new AQM scheme, CoDel [10], was proposed to control the latency directly to address the bufferbloat problem. CoDel requires per packet timestamps. Also, packets are dropped at the dequeue function after they have been enqueued for a while. Both of these requirements consume excessive processing and infrastructure resources. This consumption will make CoDel expensive to implement and operate, especially in hardware.

In this paper, we present a lightweight algorithm, PIE (Proportional Integral controller Enhanced), which combines the benefits of both RED and CoDel: easy to implement like RED while directly control latency like CoDel. Similar to RED, PIE randomly drops a packet at the onset of the congestion. The congestion detection, however, is based on the queueing latency like CoDel instead of the queue length like conventional AQM schemes such as RED. Furthermore, PIE also uses the latency moving trends: latency increasing or decreasing, to help determine congestion levels. In addition, the design parameters are self-tuning, and hence PIE is robust and optimized for various network scenarios. Our DOCSIS 3.0 simulation results show that PIE can control latency around the reference under various congestion conditions. Furthermore, it can quickly and automatically respond to network congestion changes in an agile manner.

In what follows, Section II specifies our goals of designing the latency-based AQM scheme. Section III explains the scheme in detail. Section IV presents simulation results of the proposed scheme. In Section V, we discuss the implementation cost of PIE. Section VII concludes the paper and discusses future work.

II. Design Goals

We explore a queue management framework where we aim to improve the performance of interactive and delay-sensitive applications. The design of our scheme follows a few basic criteria.

- *Low Latency Control.* We directly control queueing latency instead of controlling queue length. Queue sizes change with queue draining rates and various flows' round trip times. Delay bloat is the real issue that we need to address as it impairs real time applications. If latency can be controlled to be small, bufferbloat is not an issue. In fact, we would allow more buffers for sporadic bursts as long as the average latency is under control.

- *High Link Utilization.* We aim to achieve high link utilization. The goal of low latency shall be achieved without suffering link under-utilization or losing network efficiency. An early congestion signal could cause TCP to back off and avoid queue buildup. On the other hand, however, TCP's rate reduction could result in link under-utilization. There is a delicate balance between achieving high link utilization and low latency.

- *Simple Implementation.* The scheme should be simple to implement and easily scalable in both hardware and software. The wide adoption of RED over a variety of network devices is a testament to the power of simple random early dropping/marking. We strive to maintain similar design simplicity.

- *Guaranteed Stability and Fast Responsiveness.* The scheme should ensure system stability for various network topologies and scale well with arbitrary number streams. The system also should be agile to sudden changes in network conditions. Design parameters shall be set automatically. One only needs to set performance-related parameters such as target queue delay, but no need to set any of the design parameters.

We aim to find an algorithm that achieves the above goals. It is noted that, although important, fairness is orthogonal to the AQM design whose primary goal is to control latency for a given queue. Techniques such as Fair Queueing [11] or its approximate such as Stochastic Fair Queueing (SFQ) [12] can be combined with any AQM scheme to achieve fairness. Therefore, in this paper, we focus on controlling a queue's latency and ensuring flows' fairness is not worse than those under the standard DropTail or RED design.

III. The PIE Scheme

In the section, we describe in detail the design of PIE and its operations. As illustrated in Figure 1, our scheme comprises three simple components: a) random dropping at enqueueing; b) periodic drop probability update; c) departure rate estimation.

The following subsections describe these components in further details, and explain how they interact with each other. At the end of this section, we will discuss how the scheme can be easily augmented to precisely control bursts.

A. Random Dropping

Like most state-of-the-art AQM schemes, PIE would drop packets randomly according to a drop probability, p, that is obtained from the "drop probability calculation" component. No extra step, like timestamp insertion, is needed. The procedure

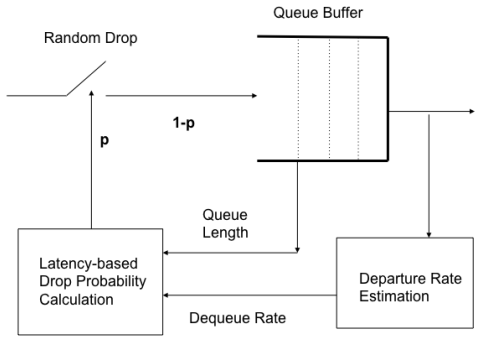


Fig. 1. Overview of the PIE Design. The scheme comprises three simple components: a) random dropping at enqueueing; b) latency based drop probability update; c) departure rate estimation.

is as follows:

Random Dropping:

Upon packet arrival

randomly drop a packet with a probability p.

B. Drop Probability Calculation

The PIE algorithm updates the drop probability periodically as follows:

- estimate current queueing delay using Little's law:
  $cur\_del = \frac{qlen}{avg\_drate}$;

- calculate drop probability p as:
  $p = p + \alpha * (cur\_del - ref\_del) + \beta * (cur\_del - old\_del)$;

- update previous delay sample as:
  $old\_del = cur\_del$.

The average draining rate of the queue, avg_drate, is obtained from the "departure rate estimation" block. Variables, cur_del and old_del, represent the current and previous estimation of the queueing delay. The reference latency value is expressed in ref_del. The update interval is denoted as T_{update}. Parameters α and β are scaling factors.

Note that the calculation of drop probability is based not only on the current estimation of the

queueing delay, but also on the direction where the delay is moving, i.e., whether the delay is getting longer or shorter. This direction can simply be measured as the difference between cur_del and old_del. Parameter α determines how the deviation of current latency from the target value affects the drop probability; β exerts the amount of additional adjustments depending on whether the latency is trending up or down. The drop probability would be stabilized when the latency is stable, i.e. cur_del equals old_del; and the value of the latency is equal to ref_del. The relative weight between α and β determines the final balance between latency offset and latency jitter. This is the classic Proportional Integral controller design [13], which has been adopted for controlling the queue length before in [7] and [14]. We adopt it here for controlling queueing latency. In addition, to further enhance the performance, we improve the design by making it auto-tuning as follows:

if $p < 1\%$: $\alpha = \tilde{\alpha}/8$; $\beta = \tilde{\beta}/8$;
else if $p < 10\%$: $\alpha = \tilde{\alpha}/2$; $\beta = \tilde{\beta}/2$;
else: $\alpha = \tilde{\alpha}$; $\beta = \tilde{\beta}$;

where $\tilde{\alpha}$ and $\tilde{\beta}$ are static configured parameters. Auto-tuning would help us not only to maintain stability but also to respond fast to sudden changes. The intuitions are the following: to avoid big swings in adjustments which often leads to instability, we would like to tune p in small increments. Suppose that p is in the range of 1%, then we would want the value of α and β to be small enough, say 0.1%, adjustment in each step. If p is in the higher range, say above 10%, then the situation would warrant a higher single step tuning, for example 1%. The procedures of drop probability calculation can be summarized as follows.

Drop Probability Calculation:

Every T_{update} interval

1. Estimation current queueing delay:

$$cur\_del = \frac{qlen}{avg\_drate}.$$

2. Based on current drop probability, p, determine suitable step scales:

$$\begin{aligned} if\ p\ <\ 1\%,\ \ \alpha =\ \ \tilde{\alpha}/8;\ \ \beta = \tilde{\beta}/8; \\ else\ if\ p\ <10\%,\ \ \alpha =\ \ \tilde{\alpha}/2;\ \ \beta = \tilde{\beta}/2; \\ else\ ,\ \ \alpha =\ \ \tilde{\alpha};\ \ \beta = \tilde{\beta}; \end{aligned}$$

3. Calculate drop probability as:

$$\begin{aligned} p = p + \alpha\ \ *\ \ (cur\_del - ref\_del) + \\ \beta\ \ *\ \ (cur\_del - old\_del); \end{aligned}$$

4. Update previous delay sample as:

$$old\_del = cur\_del.$$

We have discussed packet drops so far. The algorithm can be easily applied to networks codes where Early Congestion Notification (ECN) is enabled. The drop probability p could simply mean marking probability.

C. Departure Rate Estimation

The draining rate of a queue in the network often varies either because other queues are sharing the same link, or the link capacity fluctuates. Rate fluctuation is particularly common in wireless networks. Hence, we decide to measure the departure rate directly as follows:

Departure Rate Calculation:

Upon packet departure

1. Decide to be in a measurement cycle if:

$$qlen > dq\_threshold;$$

2. If the above is true, update departure count dq_count:

$$dq\_count = dq\_count + dq\_pktsize;$$

3. Update departure rate once $dq\_count > dq\_threshold$ and reset counters:

$$\begin{aligned} dq\_int &= now - start; \\ dq\_rate &= \frac{dq\_count}{dq\_int}; \\ avg\_drate &= (1-\varepsilon) * avg\_drate \\ &\quad + \ \varepsilon * dq\_rate \\ start &= now. \\ dq\_count &= 0; \end{aligned}$$

From time to time, short, non-persistent bursts of packets result in empty queues, this would make the measurement less accurate. Hence we only measure

the departure rate, dq_rate, when there are sufficient data in the buffer, i.e., when the queue length is over a certain threshold, $dq\_threshold$. Once this threshold is crossed, we obtain a measurement sample. The samples are exponentially averaged, with averaging parameter ε, to obtain the average dequeue rate, avg_drate. The parameter, dq_count, represents the number of bytes departed since the last measurement. The threshold is recommended to be set to 5KB assuming a typical packet size of around 1KB or 1.5KB. This threshold would allow us a long enough period, dq_int, to obtain an average draining rate but also fast enough to reflect sudden changes in the draining rate. Note that this threshold is not crucial for the system's stability.

D. Handling Bursts

The above three components form the basis of the PIE algorithm. Although we aim to control the average latency of a congested queue, the scheme should allow short term bursts to pass through the system without hurting them. We would like to discuss how PIE manages bursts in this section.

Bursts are well tolerated in the basic scheme for the following reasons: first, the drop probability is updated periodically. Any short term burst that occurs within this period could pass through without incurring extra drops as it would not trigger a new drop probability calculation. Secondly, PIE's drop probability calculation is done incrementally. A single update would only lead to a small incremental change in the probability. So if it happens that a burst does occur at the exact instant that the probability is being calculated, the incremental nature of the calculation would ensure its impact is kept small.

Nonetheless, we would like to give users a precise control of the burst. We introduce a parameter, max_burst, that is similar to the burst tolerance in the token bucket design. By default, the parameter is set to be 100ms. Users can certainly modify it according to their application scenarios. The burst allowance is added into the basic PIE design as follows:

Burst Allowance Calculation:

Upon packet arrival

1. If $burst\_allow > 0$

$$enque\ packet\ bypassing\ random\ drop;$$

Upon dq_rate update

2. Update burst allowance:

$$burst\_allow = burst\_allow - dq\_int;$$

3. if $p = 0$; and both cur_del and old_del less than $ref\_del/2$, reset $burst\_allow$,

$$burst\_allow = max\_burst;$$

The burst allowance, noted by $burst\_allow$, is initialized to max_burst. As long as $burst\_allow$ is above zero, an incoming packet will be enqueued bypassing the random drop process. Whenever dq_rate is updated, the value of $burst\_allow$ is decremented by the departure rate update period, dq_int. When the congestion goes away, defined by us as p equals to 0 and both the current and previous samples of estimated delay are less than $ref\_del/2$, we reset $burst\_allow$ to max_burst.

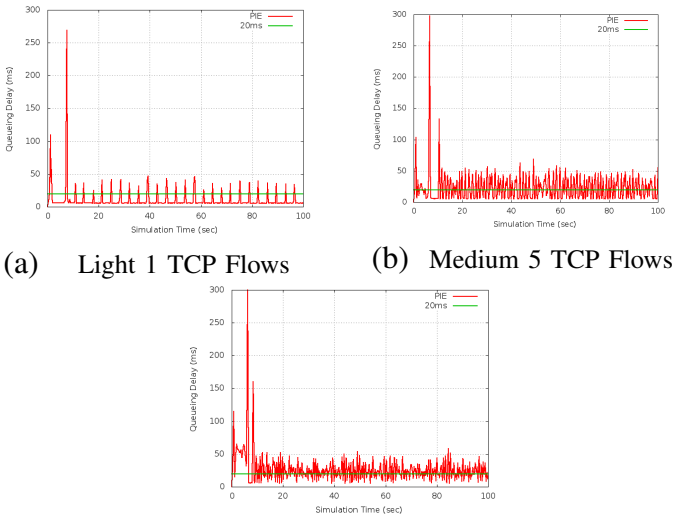
IV. Performance Evaluation

In this section we present our ns-2 [15] simulation results for the scenario of DOCSIS cable modems. We first demonstrate the basic functions of PIE using a few basic scenarios, and then compare PIE and CoDel performance using various scenarios. We focus our attention on the following performance metrics: instantaneous queueing delay, drop probability, TCP throughput and link utilization.

The CM (upstream) is modeled as a single queue that implements both rate shaping using a token bucket algorithm (with parameters Max Sustained Rate, Traffic Burst and Peak Traffic Rate), and the Request-Grant DOCSIS MAC. The CMTS (downstream) is implemented as a single rate shaping queue with token bucket. The upstream queue is modeled as a PIE (or other AQM scheme) queue, while the downstream queue is implemented using a DropTail model.

In our simulation, the upstream token bucket parameters are set to the following values: Max Sustained Rate = 5Mbps, Traffic Burst = 10MB and Peak Traffic Rate = 20Mbps. The downstream token bucket parameters are set as follows: Max Sustained Rate = 20Mbps, Traffic Burst = 20MB and Peak Traffic Rate = 50Mbps. The RTT is 100ms and unless otherwise stated the buffer size is 480ms . We

use both TCP and UDP traffic for our evaluation. All TCP traffic sources are implemented as TCP New Reno with SACK running FTP applications. UDP traffic is implemented using Constant Bit Rate (CBR) sources. Both UDP and TCP packets are configured to have a fixed size of 1500B. Unless otherwise stated the PIE parameters are configured as follow: ref_del = 20ms, T_{update} = 15ms, α = 0.25Hz, $\beta = 2.5Hz$, $dq\_threshold$ = 4500B, max_burst = 50ms.



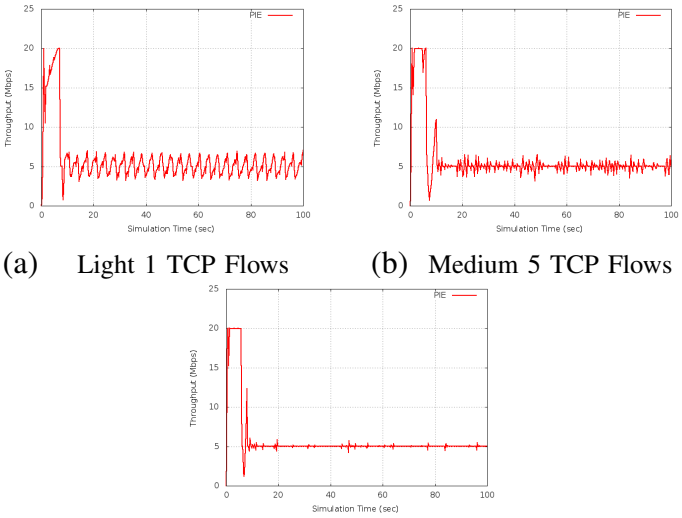
(a) Light 1 TCP Flows (b) Medium 5 TCP Flows

(c) Heavy 15 Flows

Fig. 2. Queueing Latency Under Various Traffic Loads: a) 1 TCP flow; b) 5 TCP flows; c) 15 TCP flows. Queueing latency is controlled at the reference level of 20ms regardless of the traffic intensity.

Function Verification: We first validate the functionalities of PIE, making sure it performs as designed using static traffic sources under various loads.

1) Light TCP traffic: We first consider a single TCP flow. Figure 2(a) and Figure 3(a) show the queueing delay and throughput, respectively. From Figure 3(a), it is clear that a single TCP flow is able to take advantage of the initial Peak Rate burst at 20Mbps except the initial dip around 1s. Once the token bucket transitions into the Maximum Sustained Rate of 5Mbps, we can see that the throughput oscillates around 5Mbps with the typical TCP sawtooth behavior. Due to dual token bucket rates, we are not losing throughput: if the throughput is under 5Mbps, the token of the peak rate would allow it to burst over 5Mbps. Due to the rate changes, there is a sudden spike in the queueing latency around 7s as shown in Figure 3(a). However,



(a) Light 1 TCP Flows (b) Medium 5 TCP Flows

(c) Heavy 15 TCP Flows

Fig. 3. Link Throughput Under Various Traffic Loads: a) 1 TCP flow; b) 5 TCP flows; c) 15 flows. High link utilization is achieved regardless of traffic intensity, even under low multiplexing case.

the algorithm can quickly control the delay to be around the target value of 20ms. The average drop probability here is 0.9%.

2) Medium TCP traffic: In this test scenario, we increase the number of TCP flows to 5. With higher traffic intensity, the link utilization reaches 100% for both the peak and sustained rates as clearly shown in Figure 3(b). Except the sudden spike around 7s due to the rate change, the queueing delay, depicted in Figure 2(b), is controlled around the desired 20ms. The equilibrium latency is unaffected by the increased traffic intensity. Due to higher multiplexing, the link throughput is smoother. The queueing delay fluctuates more evenly around the reference level. The average throughput reaches full capacity of 5Mbps as shown in Figure 3(b). The average drop probability is 2.0%.

3) Heavy TCP traffic: To demonstrate PIE's performance under persistent heavy congestion, we increase the traffic load to 15 TCP flows. The corresponding latency plot can be found in Figure 2(c). Again, PIE is able to contain the queueing delay around the reference level regardless of the traffic mix while achieving both the peak and sustained rates shown in Figure 3(c). The average drop probability in this case is 6.2%.

4) PowerBoost: In this test, we investigate PIE's ability to control the queuing delay when the DOCSIS model shifts to a higher upstream bandwidth:

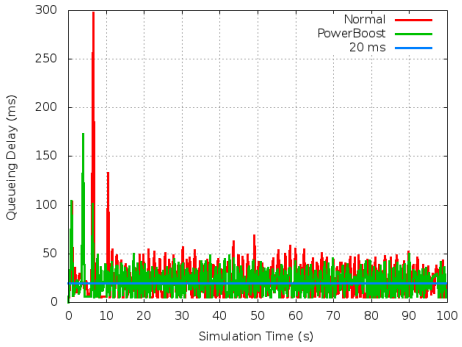


Fig. 4. Normal vs. Powerboost Delay Comparison Under 15 TCP flows: the queuing delay oscillating around the delay reference of 20ms in a similar fashion for both cases regardless of configured speeds.

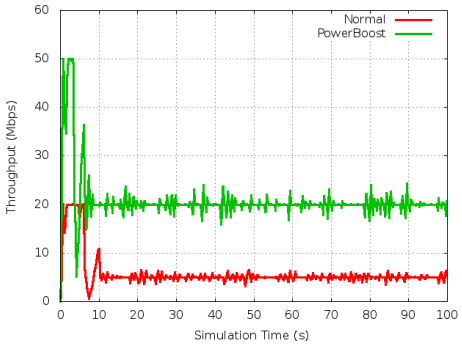


Fig. 5. Normal vs. Powerboost Throughput Comparison with 15 TCP flows: both the peak rate and the sustained rate are achieved under normal and Powerboost scenarios.

50Mbps as the peak rate and 20Mbps as the sustained rate (a.k.a PowerBoost). The critical aspect to verify is whether PIE's parameter settings hold for both the normal and PowerBoost scenarios.

Figure 4 and 5 plot the queuing delays experienced and throughputs achieved under the normal and PowerBoost conditions with 15 TCP flows. The plots show that, while different throughputs are achieved, the queuing delay oscillating around the delay reference of 20ms in a similar fashion for both cases. This verifies that PIE's auto-tuning helps PIE to adapt to the higher speed traffic conditions found in DOCSIS 3.0 cable modem environments.

Performance Evaluation and Comparison: The functions of PIE are verified above. This section evaluates PIE under various application scenarios, compares its performance against CoDel and shows

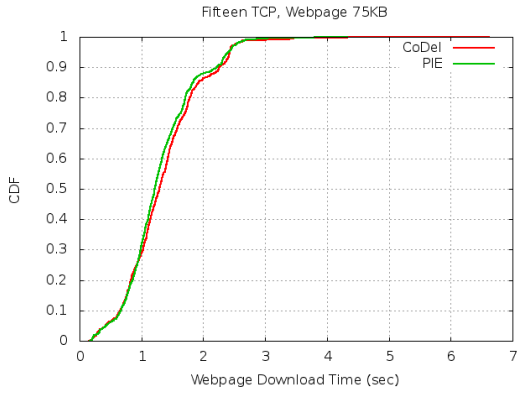


Fig. 6. PIE vs. Codel Performance Comparison: the CDF plot of 2000 web pages' download time with 15 long lived TCP flows running in the background. PIE and CoDel behave similarly in this situation.

how PIE is better suited for controlling latency in todays Internet. The simulation topology is similar to the above. The cable modem runs either the PIE or CoDel scheme.

5) HTTP Traffic: This test consists of 15 long-lived TCP flows that share the upstream bandwidth with 100 concurrent web page downloads. Each web page is 75KB in size and they are evenly downloaded from four different sites with RTTs of 20ms, 30ms, 50ms and 100ms respectively. The download process is repeated 20 times so that a total of 2000 web page downloads are generated. The long-lived TCP traffic has RTT of 100ms. Figure 6 shows the web page download time as CDF for both CoDel and PIE. From the graph, we see that the web page download times for both schemes are comparable.

6) Video traffic: This test consists of a single UDP flow at 6.5Mbps on the upstream link. Since most real time video communication adopt the UDP protocol, this test compares how each scheme behaves given a real-time, high-definition video traffic. Figure 7 shows the utilization of the upstream link for both CoDel and PIE. In both schemes, the link utilization is around 6.5Mbps (offered load) until 50s. Once tokens are exhausted at 50s, the link utilization goes down to 5Mbps for both schemes.

Figure 8 plots the queueing delay on the upstream queue. Both schemes only incur the MAC layer delay of 5ms-6ms in the initial 50s. Once the tokens are exhausted at 50s, queue builds up in both schemes. PIE is able to adapt to the increasing queue

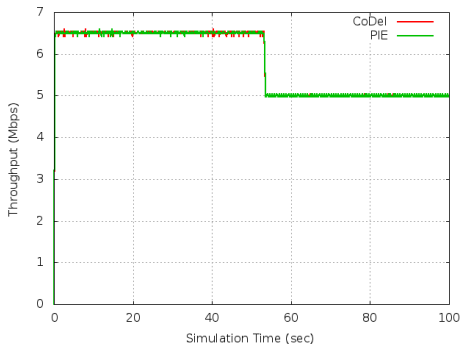


Fig. 7. PIE vs. Codel Performance Comparison Under UDP traffic: the test represents a real time video traffic which sends 6.5Mbps. In both schemes, the link utilization is around 6.5 Mbps (offered load) until 50s. Once the peak rate tokens are exhausted at 50s, the link utilization goes down to 5Mbps for both schemes.

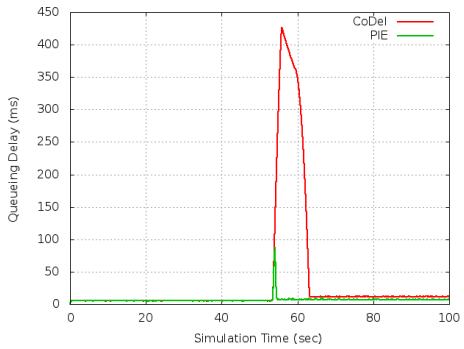


Fig. 8. PIE vs. Codel Performance Comparison Under UDP traffic: the queuing delay on the upstream queue. Both schemes maintain a low queuing delay in the initial 50s. Once the tokens are exhausted at 50s, queue builds up in both schemes. PIE is able to adapt to the increasing queue size and bring down the queuing latency around 53s, whereas CoDel takes much longer to bring down the latency (around 63s).

size and bring down the queuing latency around 53s, whereas CoDel takes much longer to bring down the latency (around 63s). PIEs auto-tuning features helps PIE to adapt faster to dynamically changing link and traffic conditions.

V. Implementation

PIE can be applied to existing hardware or software solutions. In this section, we discuss the implementation cost of the PIE algorithm. There are three steps involved in PIE as discussed in Section III. We examine their complexities as follows.

Upon packet arrival, the algorithm simply drops a packet randomly based on the drop probability p. This step is straightforward and requires no packet header examination and manipulation. Besides, since no per packet overhead, such as a timestamp, is required, there is no extra memory requirement. Furthermore, the input side of a queue is typically under software control while the output side of a queue is hardware based. Hence, a drop at enqueueing can be readily retrofitted into existing hardware or software implementations.

The drop probability calculation is done in the background and it occurs every T_{udpate} interval. Given modern high speed links, this period translates into once every tens, hundreds or even thousands of packets. Hence the calculation occurs at a much slower time scale than packet processing time, at least an order of magnitude slower. The calculation of drop probability involves multiplications using α and β. Since the algorithm is not sensitive to the values of α and β, we can choose the values, e.g. $\alpha = 0.25$ and $\beta = 2.5$ so that multiplications can be done using simple adds and shifts. As no complicated functions are required, PIE can be easily implemented in both hardware and software. The state requirement is only two variables per queue: cur_del and old_del. Hence the memory overhead is small.

In the departure rate estimation, PIE uses a counter to keep track of the number of bytes departed for the current interval. This counter is incremented per packet departure. Every T_{update}, PIE calculates latency using the departure rate, which can be implemented using a multiplication. Note that many network devices keep track an interface's departure rate. In this case, PIE might be able to reuse this information and incurs no extra cost. Besides, in cable modem or CMTS scenarios, the peak rate and sustained rate are preconfigured. Hence, PIE can take advantage of this rate information and current congestion state to simply skip the third step of the algorithm.

In summary, the state requirement for PIE is limited and computation overheads are small. Hence, PIE is simple to be implemented. In addition, since PIE does not require any user configuration, it does not impose any new cost on existing network management system solutions. SFQ can be combined with PIE to provide further improvement of latency for various flows with different priorities. However, SFQ requires extra queueing and scheduling structures. Whether the performance gain can justify the

design overhead needs to be further investigated.

VI. Acknowledgement

We would like to thank Greg White from CableLabs for providing us the ns2 model for DOCSIS 3.0 cable modems.

VII. Conclusions and Future Work

In this paper we have described PIE, a latency-based design for controlling bufferbloat in the Internet. The PIE design bases its random dropping decisions not only on current queueing delay but also on the delay moving trend. In addition, the scheme self-tunes its parameters to optimize system performance. As a result, PIE is effective across diverse range of network scenarios. Our simulation studies of DOCSIS 3.0 modems show that PIE can ensure low latency under various congestion situations. It achieves high link utilization while maintaining stability consistently. It is a light-weight, enqueueing based design that works with both TCP and UDP traffic. The PIE design only requires low speed drop probability update, so it incurs very small overhead and is simple enough to implement in both hardware and software.

Going forward, we will explore efficient methods to provide weighted fairness under PIE. There are two ways to achieve this: either via differential dropping for flows sharing a same queue or through class-based fair queueing structure where flows are queued into different queues. There are pros and cons with either approach. We will study the trade-offs between these two methods.

References

[1] B. Turner, "Has AT&T Wireless Data Congestion Been Self-Inflicted?" [Online]. Available: BroughTurnerBlog

[2] J. Gettys, "Bufferbloat: Dark buffers in the internet," *IEEE Internet Computing*, vol. 15, pp. 95–96, 2011.

[3] C. Kreibich, N. Weaver, B. Nechaev, and V. Paxon, "Netalyzer: Illuminating the edge network," in *Proceedings of Internet Measurement Conference*, 2010.

[4] G. Maier, A. Feldmann, V. Paxon, and M. Allman, "On dominant characteristics of residential broadband internet traffic," in *Proceedings of Internet Measurement Conference*, 2009.

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

[6] W. Feng, K. Shin, D. Kandlur, and D. Saha, "The blue active queue management algorithms," *IEEE/ACM Transactions on Networking*, vol. 10, no. 4, pp. 513–528, Aug. 2002.

[7] C. V. Hollot, V. Misra, D. Towsley, and W. bo Gong, "On designing improved controllers for aqm routers support," in *Proceedings of IEEE Infocom*, 2001, pp. 1726–1734.

[8] S. Kunniyur and R. Srikant, "Analysis and design of an adaptive virtual queue (avq) algorithm for active queue management," in *Proceedings of ACM SIGCOMM*, 2001, pp. 123–134.

[9] B. Braden, D. Clark, J. Crowcroft, and et. al., "Recommendations on Queue Management and Congestion Avoidance in the Internet," RFC 2309 (Proposed Standard), 1998.

[10] K. Nichols and V. Jacobson, "A Modern AQM is just one piece of the solution to bufferbloat." [Online]. Available: http://queue.acm.org/detail.cfm?id=2209336

[11] A. Demers, S. Keshav, and S. Shenker, "Analysis and simulaton of a fair queueing algorihtm," *Journal of Internetworking Research and Experience*, pp. 3–26, Oct. 1990.

[12] P. McKenney, "Stochastic fairness queueing," *Internetworking: Research and Experience*, vol. 2, pp. 113–131, Jan. 1991.

[13] G. Franklin, J. D. Powell, and A. Emami-Naeini, in *Feedback Control of Dynamic Systems*, 1995.

[14] R. Pan, B. Prabhakar, and et. al., "Data center bridging - congestion notification." [Online]. Available: http://www.ieee802.org/1/pages/802.1au.html

[15] "NS-2." [Online]. Available: http://www.isi.edu/nsnam/ns/