Auto-Tuning Active Queue Management

Joe H. Novak
University of Utah

Sneha Kumar Kasera
University of Utah

Abstract

Active queue management (AQM) algorithms preemptively drop packets to prevent unnecessary delays through a network while keeping utilization high. Many AQM ideas have been proposed, but none have been widely adopted because these rely on pre-specification or pre-tuning of parameters and thresholds that do not necessarily adapt to dynamic network conditions. We develop an AQM algorithm that relies only on network runtime measurements and a natural threshold, the knee on the delay-utilization curve. We call our AQM algorithm *Delay Utilization Knee* (DUK) based on its key characteristic of keeping the system operating at the knee of the delay-utilization curve. We implement and evaluate DUK in the Linux kernel in a testbed, that we build, and in the ns-3 network simulator. We find that DUK can attain reduced queuing delay and reduced flow completion times compared to other algorithms with virtually no reduction in link utilization under varying network conditions.

I. Introduction

Active Queue Management (AQM) tries to answer the question, "What is the best way to preemptively drop packets to prevent unnecessary delays through a network while keeping utilization high?" Many ideas have been proposed to answer this question[1], yet despite the potential benefits of AQM, none of the ideas have been widely adopted in routers and other network elements. The key problem is that the existing ideas require pre-specification of AQM parameters and thresholds (e.g., RED [5], CoDel[2] [2].) Given the dynamic nature of networks due to changing traffic patterns and also due to changes in link capacity or bandwidth, finding the right values of the AQM parameters and thresholds is very hard. Moreover, one cannot capture all network situations with the same set of parameters. Another well-known existing work, PIE [3], uses a pre-tuning approach where the AQM parameters are experimentally determined for the network element and the environment in which the AQM would be deployed. This approach also fails to capture wide-scale changes in the network conditions. In this paper, we tackle the challenge of designing an AQM method that is based on the principle that no parameters should be pre-specified or pre-tuned. Instead, we look for a threshold that occurs naturally in queuing systems. Our AQM approach does not require pre-specification of even those parameters including EWMA[3] weights or protocol timer intervals that are routinely specified in protocol design and implementation.

When we view the network link in terms of utilization and the queue corresponding to the link in terms of delay (Fig. 1 shows a typical delay-utilization curve), we see that as utilization increases, delay also increases. At a certain point, however, there is a very large increase in delay for only a small improvement in utilization. This disproportionate increase in delay is of little to no value to the applications at the endpoints. We want to avoid this *unstable* region of high increase in delay with little increase in utilization. As a result, a natural threshold becomes apparent. This point at which the change in delay becomes greater than the change in utilization is the knee on the curve. Rather than pre-specified queue length thresholds or delay parameters, we use this natural threshold to derive an expression for the AQM packet drop or marking probability.[4] We recognize that the delay-utilization curve is valid over large time scales. However, networks operate over short time scales. We do not have the luxury of taking measurements or reacting over a long time. Conceptually, we base our approach on the delay-utilization curve, but implement it at short time scales.

Our AQM algorithm adapts to network environments without the need for tuning or adjustment of thresholds. Very importantly, it does not require any buffer sizing, i.e., it does not require setting limits on buffer sizes and hence our AQM mechanism does not depend on tail drops. We call our AQM algorithm *Delay Utilization Knee* (DUK in short) based on its key characteristic of keeping the system operating at the naturally occurring knee of the delay-utilization curve. The primary goal of DUK is to quickly adapt to any bandwidth or offered load without reducing link utilization or unnecessarily increasing queuing delay. We implement and evaluate DUK in the Linux kernel in a testbed, that we build, and in the ns-3 simulator [7] under a variety of network operating conditions. Our results indicate we can obtain robustness to changing and differing network environments without sacrificing performance. Very importantly, we demonstrate that with ECN, our approach decreases flow completion times compared to other algorithms. We compare DUK with two contemporary algorithms, CoDel and PIE, and find that DUK can attain decreased queuing delay with virtually no reduction in the link utilization. We choose to compare DUK to CoDel because it claims to have no parameters or thresholds to tune to the network environment.

[1]Recently, it has been discussed in the context of the continuing or even worsening bufferbloat problem [1], [2], [3], [4]. Bufferbloat is the oversizing of buffers in network devices which can lead to excessive delays without any real benefit to network providers.

[2]Contrary to its claims, CoDel requires specification of parameters including a minimum queuing delay.

[3]Exponentially Weighted Moving Average.

[4]AQM algorithms probabilistically drop packets or mark them using Explicit Congestion Notification (ECN) [6] when they detect congestion.

We compare DUK to PIE to demonstrate the advantages of a self-adaptive algorithm compared to one that needs to be pre-tuned and may not react well when network conditions change. Thus, DUK offers a highly robust alternative to existing AQM methods and can be widely deployed without worrying about pre-selection or pre-tuning of AQM parameters.

II. DUK Goals and Approach

The key goals in designing DUK are as follows. First, DUK should not depend upon any pre-tuning or pre-specifcation of parameters but only on runtime measurements. Second, DUK should not make any traffic source behaviour assumptions. The traffic at a network node can be a mix of different flavors of TCP, multimedia traffic or other types of traffic with their own congestion control mechanisms, etc. Thus, determining parameters based on any particular flavor of TCP is not likely to be optimal across different traffic types. In designing DUK, we do not necessarily aim for optimality. Unlike some of the existing work on self-tuning AQM [8], we do not make any TCP assumptions. DUK adapts its drop or marking probability using local measurements only. Third, DUK does not require any per-connection information such as per-connection round trip times (RTTs) required by some existing approaches [3], [9]. Moreover, per-connection information is also not necessarily available when traffic is encrypted (e.g., when using IPsec).

Dependence on a Natural Threshold

Given that the primary goal of an AQM algorithm is to balance the queuing delay with the link utilization, we choose the knee on the delay-utilization curve as the desired natural threshold. At this point, the first derivative of the delay D with respect to the utilization U is equal to unity. We label this point $dD/dU = 1$ in Fig. 1. We want to avoid the region that lies to the right of this point because this is where the delay increases more quickly than the utilization and makes the queueing system unstable. One could possibly argue choosing dD/dU slightly greater than 1 or possibly less than 1. Our choice maximizes utilization while keeping the delays low without any flirtation with the unstable region.

The tangent at the knee is always 1 regardless of the time units we use for delay. As the granularity of runtime measurements becomes finer, the measured change in utilization decreases proportionally. Thus, the knee location naturally scales to the time units. For example, if we measure the change in delay over 1 second, the change in utilization will be proportionally higher than if we measure the change in delay over 10 ms. Although the delay-utilization curve is valid for long time scales, we base our approach on this concept. In practice, we must react over small time scales. We choose time units in terms of the resolution of the system clock for greatest accuracy. The time interval algorithm described later ensures that the time units are appropriate for the network environment.

III. AQM Probability Determination

In this section, we derive an expression for the probability of dropping or marking packets with ECN using the natural threshold $dD/dU = 1$ for a router queuing system comprising a FIFO queue and a link with a certain bandwidth. DUK does not need a priori knowledge of the link bandwidth. Let $\frac{dD}{dU}\mid_{U_i}$ be the point where the queuing system is initially operating. The utilization at this point is U_i. Let $\frac{dD}{dU}\mid_{U_f}$ be the point that the queuing system reaches after some packets are admitted or dropped (or marked). The utilization at this point is U_f. Let U_{i-1} be the utilization during the previous evaluation of the algorithm.

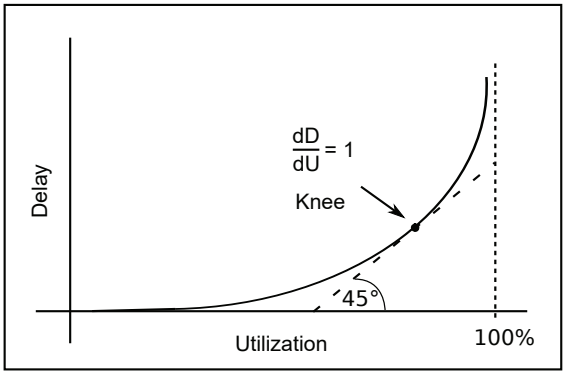


Fig. 1: Delay-Utilization Graph

In order to determine the packet drop or marking probability, we make the simplifying assumption that the second derivative of the delay-utilization curve is constant over a short time. Note that we neither control nor enforce this assumption in our experiments. Under this assumption,

$$\frac{dD}{dU}\mid_{U_f} = \frac{dD}{dU}\mid_{U_i} + \frac{d^2D}{dU^2}(U_f - U_i) \tag{1}$$

DUK calculates $\frac{dD}{dU}\mid_{U_i}$ from measurements taken at run time. Given our goal to reach $\frac{dD}{dU}\mid_{U_f} = 1$, we have:

$$1 = \frac{dD}{dU}\mid_{U_i} + \frac{d^2D}{dU^2}(U_f - U_i) \tag{2}$$

We derive an expression for $\frac{d^2D}{dU^2}(U_f - U_i)$ as follows. We compute the change in delay dD over a short time dt as the difference between the admit rate a and the transmit rate β divided by the capacity C as shown in Equation 3. We assume the arrival rate A and the capacity C remain constant over dt. We do not control these assumptions in our experiments.

$$\frac{dD}{dt} = \frac{a - \beta}{C} = \frac{a - UC}{C} \tag{3}$$

Here, U is the link utilization. The admit rate is equal to the probability of admitting or keeping a packet p_k multiplied by the arrival rate.

$$a = p_k A \tag{4}$$

Using Equation 3 and the chain rule we obtain:

$$\frac{dD}{dU} = \frac{dD}{dt}\frac{dt}{dU} = \frac{a - UC}{C} / \frac{dU}{dt} \tag{5}$$

Algorithm 1 DUK Algorithm

```
// section 1-initialize and check idle
A = arrival rate
C = estimate of capacity
k = queue length in packets
dD/dU |_{U_i} = current d(delay) / d(utilization)
Δt = estimate of measurement time interval
U_i − U_{i−1} = change in utilization
if (k == 0 or A == 0){ p = 0; return }
// section 2
Compute p as per Equation 9
if (p < 0) { p = 0 } else if (p > 1) { p = 1 }
```

Algorithm 2 Time Interval Algorithm

```
//P_{last|cur} = previous or current congestion
//level as per Equation 10
//β_{old|cur|new} = old, current, or new estimate of
//transmission rate
//Δt_{old|cur|new} = old, current, or new measurement
//time interval
if (P_cur != P_last){
    if (β_cur>=β_old at P_cur and Δt_cur<Δt_old at P_cur{
        β_new at P_cur = β_cur
        Δt_new at P_cur = Δt_cur}
    Set m and c from curve fit of Equation 11}
Δt_new = mP_cur + c
```

We differentiate Equation 5 with respect to time to obtain the second derivative as shown in Equation 6.

$$\frac{d^2D}{dU^2} = -1/(\frac{dU}{dt}) - \frac{a - UC}{C}\frac{d^2U}{dt^2}/(\frac{dU}{dt})^3 \tag{6}$$

To reach our desired point on the delay-utilization curve, we find the second derivative at $U = U_f$. Over a short time interval Δt, we approximate dU/dt by $(U_f - U_i)/\Delta t$, and d^2U/dt^2 by $((U_f - U_i) - (U_i - U_{i-1}))/\Delta t^2$. We plug these into Equation 6 to obtain the following:

$$\frac{d^2D}{dU^2}(U_f - U_i) = -(1 + \Delta t) + \frac{U_i - U_{i-1}}{U_f - U_i} \tag{7}$$

At our reference point, $dD = dU$, we have $U_f - U_i/\Delta t = (a - U_f C)/C$. We also find $U_f - U_i$ to be equal to $\Delta t(\frac{a}{C} - U_i)/(1 + \Delta t)$ and use Equations 7 and 2 to obtain Equation 8.

$$1 = \frac{dD}{dU}\,|_{U_i} -(1 + \Delta t) + \frac{(U_i - U_{i-1})(1 + \Delta t)}{\Delta t(\frac{a}{C} - U_i)} \tag{8}$$

Finally, we plug Equation 4 into Equation 8 and solve for p_k. We obtain the drop or marking probability $p = 1 - p_k$ from Equation 9 on which we base the DUK algorithm.

$$p = 1 - \frac{C}{A}\left[\frac{(U_i - U_{i-1})(1 + \Delta t)}{(1 - \frac{dD}{dU}\,|_{U_i} + (1 + \Delta t))\Delta t} + U_i\right] \tag{9}$$

The various parameters in the above equation can be computed very efficiently with minimal overhead as we discuss in the next section. Equation 9 can evaluate to values outside the range [0, 1] during abrupt changes in delay or utilization. We account for this in the DUK Algorithm described below.

IV. DUK Algorithm

We show the DUK algorithm in Algorithm 1. The router queuing system executes the algorithm on a periodic basis. The algorithm computes a drop or marking probability for packets received over the subsequent period. The device then drops or marks the packets with the probability determined by the algorithm. If a packet is not dropped, the device accepts it and appends it to the tail of the transmission queue.

The algorithm consists of two sections. The first initializes variables and detects idle link conditions. Here, DUK sets the drop or marking probability to zero any time the queue becomes empty or when no data is being received. We measure all values used in Equation 9. The utilization is the transmission rate divided by the estimate of the link capacity $U = \frac{\beta}{C}$. We develop an efficient algorithm for estimating C; however, we omit the description due to a lack of space.

The second section computes the drop or marking probability according to Equation 9. It is possible for this equation to return a value outside of [0, 1] when abrupt changes in the delay-utilization curve occur. Our algorithm checks for this condition and truncates the probability to valid limits.

Interval Determination: The measurement time interval Δt between iterations of the algorithm can affect stability. Existing AQM approaches use a non-adaptive, arbitrarily specified value for this interval. Following the primary theme of this paper, we use a measurement-based self-adaptive approach for determining Δt, as we describe below and show in Algorithm 2, that does not rely on pre-specifications.

We express the backlog or congestion at the network queue as a percentage as shown in Equation 10 where P is the percentage of congestion, M is the estimate of the ingress link capacity and C is the estimate of the egress link capacity.

$$P = 100 * (M - C)/M \tag{10}$$

We collect triples $(P_i, \Delta t_i, \beta_i)$ of congestion level P_i, time interval samples Δt_i, and transmission rate β_i as the network environment changes. We want to find the minimum Δt_i for the current P_i. For each P_i we encounter, we update Δt_i and β_i if the current transmission rate is greater than or equal to any previous sample and the current time interval is less than any previous sample. Effectively, we keep the smallest time interval for which the transmit rate does not decrease. When we encounter a new congestion level P_i, we do not have an initial value for Δt_i. To remedy this, we fit a line (Equation 11) to the data points we have collected so far and use this as the initial value for the new congestion level. In Equation 11, m is the slope and c is the vertical axis intercept (corresponding to the time interval when there is no congestion).

$$\Delta t = mP + c \tag{11}$$

We use Algorithm 2 to determine the next time interval Δt_{new}. We note the slope of the line in Equation 11 is expected

to be negative because of the following intuition: the higher the congestion level, the faster the AQM algorithm should react, and correspondingly, the lower should be the value of Δt.

Implementation: We note that the computational overhead of DUK's probability calculation is minimal because it is evaluated only once per time interval. However, to minimize computation cost, we use scaled integer rather than floating point arithmetic. We also reduce the number of divisions required by simplifying the ratio of C/A. DUK keeps track of the number of bytes of the numerator and denominator over the same time interval and performs a single division to obtain the ratio. With these simplifications, there are three divisions (one for each of dD/dU, C/A, and the primary term over Δt) and three multiplications in the probability calculation. By comparison, PIE's probability calculation involves one division and two multiplications. Our commodity Linux box can evaluate DUK's probability 25 million times per second. At this rate, the minimum time interval is about 40 nanoseconds which is more than sufficient for today's highest network speeds.

PIE estimates the current queuing delay by dividing the queue length by the average departure rate thus requiring an EWMA parameter. We instead use a technique similar to CoDel which uses timestamps to determine the queuing delay. We run the interval determination algorithm as a low priority background task that refines the fitted line. It is not in the critical path and does not impact performance.

V. Experimental Setup

We implement DUK in the Linux kernel on physical hardware in a testbed that we build. We extensively compare its performance with CoDel and PIE. Before we describe our setup, we briefly review CoDel and PIE and their parameters.

CoDel: The CoDel algorithm measures the sojourn time each packet in the queue. CoDel always admits packets to the queue and makes a drop or marking decision each time a packet is removed from the queue based on the measured sojourn time. If the sojourn is above a target time of 5 ms for a period of at least 100 ms, CoDel enters a drop/mark state. While in this state, CoDel drops or marks packets with exponentially increasing frequency until the sojourn falls below 5 ms.

PIE: PIE is based on a feedback system that uses various parameters and thresholds to control its estimates and calculations. We use the default values for these parameters and thresholds. When a packet is dequeued, PIE estimates the queuing delay from the queue length and an EWMA of the dequeue rate. Using PIE's notation, every update period of $T_{update} = 30\,ms$, it increments or decrements the probability of dropping or marking a packet by adding two weighted factors. The first is the difference between the delay estimate and a reference delay. The weight for this factor is $\tilde{\alpha} = 0.125\,Hz$ and the reference delay is $delay\_ref = 20\,ms$. The second is the difference between the current and previous delay estimates. The weight for this factor is $\tilde{\beta} = 1.25\,Hz$. Estimates are updated only if the queue length is greater than a threshold of $dq\_threshold = 10\,KBytes$. PIE uses a token bucket design that allows packets to be excluded from the drop or marking

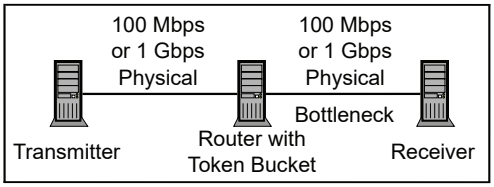


Fig. 2: Configuration 1 - Single Transmitter

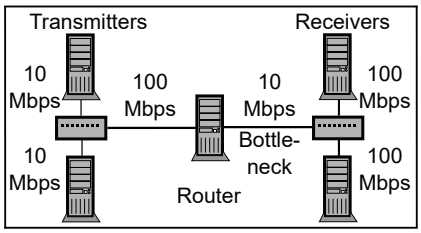


Fig. 3: Configuration 2 - Dual Transmitters

decision during an initial congestion transient. The duration of this is controlled by the parameter $max\_burst = 100\,ms$.

We base our CoDel and PIE modules on their respective kernel [10] and reference implementations [11]. Additionally, we simulate DUK in ns-3 to conduct large scale experiments. We use the ns-3 CoDel model [12]. We port PIE to ns-3.

We run DUK, CoDel, and PIE in each experiment for comparison. To compare the performance of the base algorithms, we eliminate the maximum queue limits at which CoDel and PIE revert to DropTail. We add a variable RTT component to the network to mitigate synchronization effects.

Physical Hardware: We run Linux Kernel 3.9.10 with TCP CUBIC in our testbed. Our modules are Linux *qdisc*[5] modules [13]. We use two network configurations for our experiments. The first consists of a single transmitter, a router, and a receiver (Figure 2). We set the physical link speeds to either 100 Mbps or 1 Gbps depending on the experiment. We use a token bucket to create a bottleneck between the router and receiver. We program the token bucket to run a bandwidth schedule specific to the experiment. In the second configuration (Fig. 3), we add an additional transmitter and receiver. We connect the transmitters to the router through a link layer switch. We connect the receivers to the router through a separate link layer switch. This configuration ensures the queuing algorithms in the switches do not interfere with the results. The link rates shown create a physical bottleneck between the router and the switch on the receiver side. We do not use a token bucket in this configuration.

We use commodity Intel-based hardware for all machines. The router is a 3 GHz Core i5-2320. In Fig. 2, the transmitter is a 3.4 GHz i7-4770 and the receiver is a 2.26 GHz Core i5 430M. For diversity (Fig. 3), we add a 2 GHz Pentium 4 transmitter and a 1.6 GHz Atom N2600 laptop receiver.

We use TCP flows unless otherwise stated. We use the Linux *netem qdisc delay* [14] to add delay to the RTT. This delay

[5]A *qdisc* (queuing discipline) is a network scheduler that is part of the network traffic control subsystem in the Linux operating system.

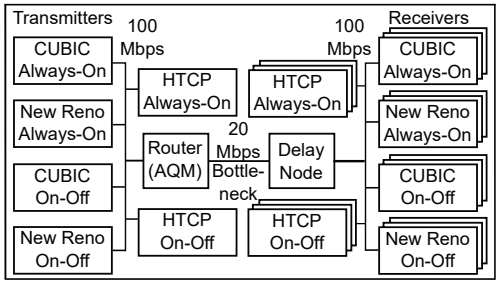


Fig. 4: Large Scale Configuration

simulates a link level delay and affects all traffic on the link. We set the *netem* buffer size high enough that it does not interfere with the experiments. The *netem* program takes *delay* and *jitter* as parameters. Delay is normally distributed and the *delay* parameter is the mean of the distribution. Unless otherwise stated, we use a *delay* of 50 ms and a *jitter* of 10 ms.

We set the packet size to 1448 bytes. We use a custom TCP application that simulates bulk transfer of a large data stream such as file transfer or video streaming. In some scenarios, we create short-lived flow transmitters to simulate web browsing. When a short-lived transmitters finishes sending a flow, it creates another to take its place. The application sets the TCP buffer sizes as large as the operating system allows to minimize effects of host limitations. We use *iperf* to generate UDP traffic.

Simulation: We implement DUK in ns-3 version 3.19 to create large scale environments. We port PIE to ns-3 and we add wrapper code to CoDel that allows it to interface to our statistics module. We use the Network Simulation Cradle (NSC)[6] to run various TCP stacks. We load New Reno, CUBIC, and HTCP to create a heterogeneous environment. We create transmitters that continually send data to simulate large file transfers or video streaming. We create short-lived flow transmitters to simulate web browsing. When a short-lived flow completes, the sender creates another to take its place. We do not run ECN in ns-3 because it is not supported with NSC.

Fig. 4 shows our ns-3 network topology. We create this topology with the transmitters and receivers separated by router and delay nodes. We run the AQM algorithms on the router. We modify the BridgeNetDevice ns-3 class to create a delay node in which we can configure the RTT and jitter. We set the delay to 50 ms and the jitter to 10 ms. We set the transmitter and receiver sides of the network to 100 Mbps. We create a bottleneck by setting the router's egress link to 20 Mbps.

VI. Results

There is little variation between runs of the same experiment for the same algorithm. Nevertheless, we run each experiment 5 times and report averages. We evaluate the following metrics: (i) *Queuing delay:* is the amount of time a packet spends in the queue and being served. (ii) *Queuing delay standard deviation:* shows the queuing delay variation over the the experiment and quantifies jitter. (iii) *Queuing delay interquartile range (IQR):* is a measure of the statistical dispersion of the queuing delay and quantifies stability. (iv) *Utilization:* is the percentage of the available capacity used by the system. (v) *Drops/marks:* is the percentage of packets dropped or marked. (vi) *Completion time:* is the amount of time for flows to complete. In some cases, we report third quartile completion times for the short-lived flows.

[6]NSC allows the simulator to run on a real world TCP stack by loading a Linux TCP stack. We use nsc 0.5.3 with liblinux2.6.18.

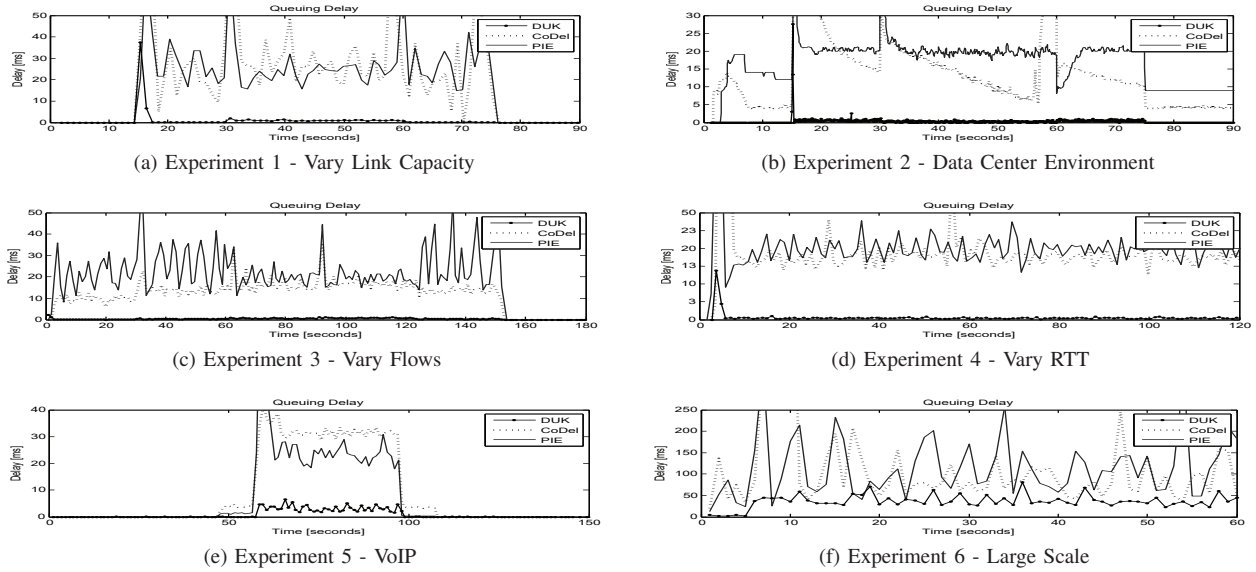
The importance of ECN

We would like to make an important point about ECN before describing our experimental results. We believe that ECN is important to the successful deployment of AQM systems. As our drop experiments show, DUK is able to control the queue length much better than competing mechanisms; however, that comes at the cost of the increased drop probability. This is not surprising, as with TCP flows a decrease in queuing delay induces a quadratic increase in drop or probability [9]. Using the well known "square-root p" formula of TCP throughput, if we have N flows passing through a congested link, then we have $C = \sqrt{2/p}N/R$ where p is the drop probability and R is the round trip delay. If the AQM mechanism is successfully able to reduce the delay to R_{aqm}, we have $C = \sqrt{2/p}N/R = \sqrt{2/p_{aqm}}N/R_{aqm}$. This implies $p_{aqm} = (R/R_{aqm})^2 p$. Thus, any gains from reducing the queuing delay are lost (quadratically) in the loss rate. With higher loss rate comes a greater potential to cause timeouts and to introduce application jitter, especially in shorter flows. However, if we use *ECN* instead of drops, p_{aqm} is simply a feedback signal and we can get lower latency at zero cost. This logic applies to all AQM mechanisms and hence it is our strong belief that AQM works best with ECN. Since ECN is widely deployed now, time is ripe for successful deployments of AQM as well.

We now present our results. We refer to Fig. 2 as configuration 1 and Fig. 3 as configuration 2. We summarize drop experiments in Table I and ECN in Table II. We show results after initial transients have settled. We show graphs of sample runs in Figs. 5 and 6 for drops and ECN, respectively. We average the data once per second for clarity at the expense of resolution. This obscures oscillations; however, the standard deviation and IQR indicate the severity. Note that in each graph, DUK has the least amount of queuing delay and is the lowest plot in each figure. DUK's queueing delay in most cases is less than half the queuing delay of either CoDel or PIE. In the VoIP experiment, it is about one tenth of the other algorithms.

Experiment 1 - Vary Link Capacity: We use configuration 1 at 100 Mbps. Initially, the token bucket sets the capacity to 100 Mbps. We create 40 short-lived flows of 250 KB each. At 15 seconds, the token bucket sets the capacity to 50 Mbps, 20 Mbps at 30 seconds, 50 Mbps at 60 seconds, and 100 Mbps at 75 seconds. We end the experiment at 90 seconds.

The tables show metrics during the most severe congestion which occurs at 20 Mbps capacity. We show sample runs in Figs. 5(a) and 6(a) averaged over two-second intervals for clarity. DUK exhibits a 10x improvement in queuing delay, standard deviation, and IQR in the drop experiment. We see that the third quartile ECN flow completion times for DUK



(a) Experiment 1 - Vary Link Capacity

(b) Experiment 2 - Data Center Environment

(c) Experiment 3 - Vary Flows

(d) Experiment 4 - Vary RTT

(e) Experiment 5 - VoIP

(f) Experiment 6 - Large Scale

Fig. 5: Queuing Delay, Drop Packets

TABLE I: Experiment Summary Data, Drop Packets

Experiment	Algorithm	Mean Queuing Delay (ms)	Std. Dev. of Queuing Delay (ms)	IQR of Queuing Delay (ms)	Utilization (%)	Packet Drops (%)
1 - Vary Capacity	DUK	1.00	1.16	1.00	98.5	16.96
	CoDel	30.76	11.99	18.00	99.7	9.73
	PIE	25.82	9.18	13.00	99.8	9.86
2 - Vary Capacity, DC	DUK	0.19	0.42	0.00	99.9	24.33
	CoDel	14.10	3.86	7.00	99.9	7.55
	PIE	18.85	0.90	1.00	100.0	5.91
3 - Vary Flows	DUK	0.80	0.98	1.00	99.9	15.19
	CoDel	17.62	6.06	8.00	100.0	6.41
	PIE	19.50	6.02	8.00	100.0	5.73
4 - Vary RTT	DUK	0.47	0.84	1.00	99.2	1.98
	CoDel	12.20	6.38	8.00	99.7	0.96
	PIE	24.77	5.94	8.00	99.8	0.60
5 - VoIP	DUK	3.43	4.29	5.00	99.6	14.92
	CoDel	30.52	10.04	15.00	100.0	8.44
	PIE	27.50	9.68	14.00	100.0	7.27
6 - Large Scale	DUK	37.73	31.89	45.55	93.0	25.81
	CoDel	94.42	78.35	106.80	93.1	24.60
	PIE	112.65	105.63	145.88	93.3	11.24

are about 1.25 seconds shorter than PIE or CoDel. This improvement comes from the shorter queuing delay.

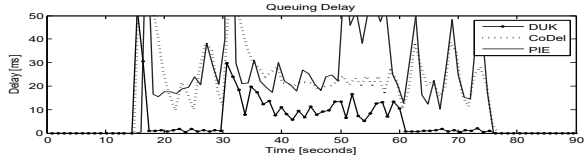
Experiment 2 - Vary Link Capacity, Data Center Environment: This is similar to Experiment 1, but at higher link capacity. Figs. 5(b) and 6(b) show sample runs. Unfortunately, testbeds that operate at higher data rates, that are typical in data center networks, that would also allow changes in the queue management, are not available to us. We use a 1 Gbps link rate, the maximum supported by our hardware, in configuration 1 to approximate a data center environment. We remove the *netem* link delay to simulate the low RTTs typical of data centers [15]. Initially, the token bucket sets the capacity to 1 Gbps. The short-lived flow sender creates 100 flows of 2500 KB each. At 15 seconds, the token bucket sets the capacity to 500 Mbps, 200 Mbps at 30 seconds, 500 Mbps at 60 seconds, and 1 Gbps at 75 seconds. We run the experiment for 90 seconds.

The tables show data for the 200 Mbps interval. Utilization is nearly identical, but DUK significantly outperforms in queuing delay. It also improves completion times. In a real data center, we expect the timer interval to be less than 1 ms; however, we are limited by the resolution of our Linux implementation. With a higher precision timer, DUK can attain lower delays.

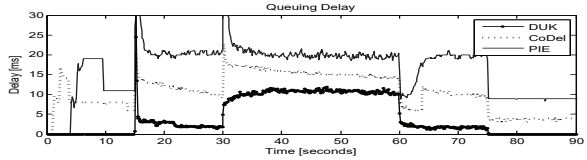
Experiment 3 - Vary Flows: Figs. 5(c) and 6(c) show sample runs. We evaluate the response to changes in offered load using configuration 2. We begin with 5 bulk TCP flows between the first sender-receiver pair. At 30 seconds, we add 5

TABLE II: Experiment Summary Data, Mark Packets with ECN

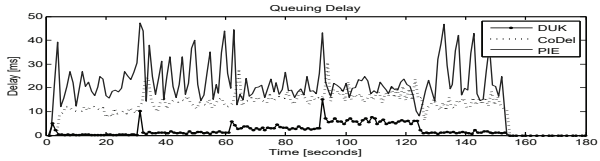
Experiment	Algorithm	Mean Queuing Delay (ms)	Std. Dev. of Queuing Delay (ms)	IQR of Queuing Delay (ms)	Utilization (%)	Packet Marks (%)	Third Quartile Completion Time (seconds)
1 - Vary Capacity	DUK	11.86	6.67	9.00	99.9	38.54	6.17
	CoDel	24.21	7.07	10.00	99.9	30.67	7.43
	PIE	28.07	17.87	23.00	99.8	37.12	7.43
2 - Vary Capacity, DC	DUK	11.05	0.55	0.00	100.0	39.13	9.49
	CoDel	17.13	0.74	1.00	100.0	13.58	11.96
	PIE	19.40	0.72	1.00	100.0	9.92	12.05
3 - Vary Flows	DUK	5.70	5.38	9.00	99.8	19.28	N/A
	CoDel	17.57	6.14	8.00	100.0	9.39	N/A
	PIE	20.50	5.59	8.00	100.0	8.11	N/A
4 - Vary RTT	DUK	0.67	1.34	1.00	98.5	5.67	N/A
	CoDel	11.33	5.03	7.00	100.0	0.00	N/A
	PIE	24.26	7.25	10.00	100.0	0.00	N/A



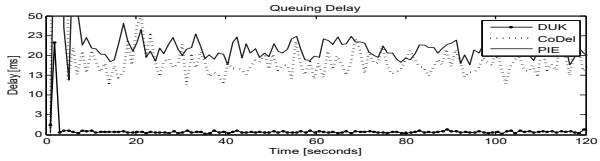
(a) Experiment 1 - Vary Link Capacity



(b) Experiment 2 - Data Center Environment



(c) Experiment 3 - Vary Flows



(d) Experiment 4 - Vary RTT

Fig. 6: Queuing Delay, Mark Packets with ECN

flows between the second pair. At 60 seconds, we add 5 flows of cross traffic from the first sender to the second receiver. At 90 seconds, we add 5 flows from the second sender to the first receiver. At 120 and 150 seconds, we remove 5 flows from each sender. The tables show data for the 20-flow time. DUK's queuing delay is 20x lower for drops and 3x lower for ECN.

Experiment 4 - Vary Round Trip Time: We set the RTT between the first sender-receiver pair to 25 ms with a jitter of 5 ms. We set the RTT between the second pair to 100 ms with a jitter of 15 ms. We run 3 bulk flows between each pair for 120 seconds. We show sample runs in Figs. 5(d) and 6(d).

With a light load, PIE becomes unstable. This in line with the prediction in [16] which states that AQM systems become more unstable with fewer flows and higher RTTs. A principled approach to make the PI controller self-tuning has been presented in [8] but PIE is not self-adaptive in that sense. As a result, PIE begins to oscillate and introduces significant jitter. With ECN, CoDel and PIE mark only a few packets. DUK is able to maintain a consistent and low queuing delay.

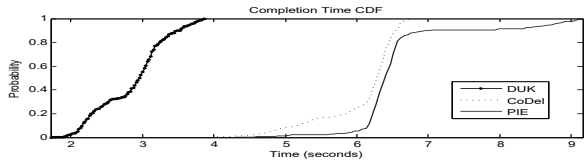
Experiment 5 - VoIP: We evaluate DUK in the presence of unresponsive UDP flows. Note that ECN marking is not an option with UDP. We simulate a small VoIP network with 5 Mbps dedicated to voice calls. Each call is 87.2 Kbps [17] and the link saturates with 58 simultaneous calls. Using configuration 1, we add 10 UDP flows every 10 seconds for 60 seconds. Beginning at 90 seconds, we remove 10 UDP flows every 10 seconds. Fig. 5(e) shows a sample run. We compute metrics for the saturated time. DUK shows a 10x lower queuing delay.

Experiment 6 - Large Scale Heterogeneous Network Simulation: We use ns-3 to evaluate DUK in a large scale network with a mix of TCP implementations. We create one always-on sender of each TCP variant. We connect three receivers to each sender for a total of 9 bulk flows to simulate video streaming or large file transfers. We create one short-lived flow sender of each TCP variant and connect 9 receivers to each. Each pair runs 50 simultaneous connections (1350 total) to simulate web traffic. In total, there are 44 nodes and 1809 connections. This experiment illustrates DUK's ability to scale to large networks. DUK shows 2x-3x improvement in delay, IQR, and standard deviation. Fig. 5(f) shows a sample run.

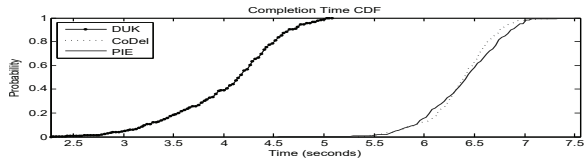
With IQR, a lower number indicates higher stability. In nearly every case, DUK's IQR is lower than the other algorithms and in many cases significantly lower. It is 1/3 lower for VoIP and 1/2 lower for large scale simulation.

Flow Completion Times: We perform additional ECN experiments to investigate flow completion times. Because of its shorter queuing lengths, DUK's times are shorter. We illustrate with CDFs (Cumulative Distribution Functions).

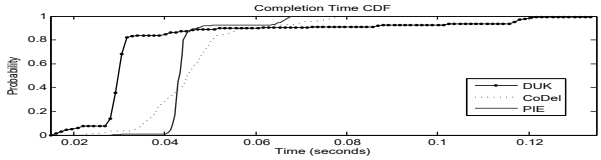
In the first experiment, we use configuration 1 at 100 Mbps with a 20 Mbps bottleneck. We run 25 senders that continually



(a) 25 ms RTT



(b) 50 ms RTT



(c) 1 Gbps Link Speed

Fig. 7: Completion Time CDFs

send short-lived flows of 250 KB each. We observe in Figs. 7(a) and 7(b) that improvement is dependent on RTT. The third quartile of DUK's completion times is nearly 50% lower with 25 ms RTT and 33% lower with 50 ms RTT.

In the second experiment, we use configuration 1 at 1 Gbps with a 500 Mbps bottleneck. We remove the delay node. We run 250 bulk flows background traffic. We run 50 senders that continually send short-lived flows of 25 KB each. The third quartile shows a 30% to 40% reduction in time.

Fairness: We compute Jain's Fairness Index [18] for experiments with long-lived TCP flows. In all experiments, all algorithms exhibit an index within 0.5% of each other.

VII. Related Work

Many existing AQM approaches [5], [19], [20], [21], [22] are hard to configure and require specification of parameters. More recent algorithms such as CoDel and PIE focus on queuing delay. Hollot et al. [9] use classical control theory to develop the PI (Proportional-Integral) controller. PIE is based on this work. PIE is a linear feedback control system with thresholds of queuing delay and maximum queue length. Its parameters include deviation from a target queue delay and a balance between queuing delay deviation and jitter. Hong et al. [23] set a probability to maintain a reference queue length which is an arbitrarily chosen constant tuned to the network. Our work differs in that we set the drop probability to maintain a naturally occurring position on the delay-utilization curve. Chiu and Jain [24] define the knee of a throughput versus offered load curve. In [25], Jain presents CARD TCP end-point congestion avoidance. It uses the gradient of the RTT vs. TCP window size curve to determine if the TCP window size should be increased or decreased. We implement an AQM algorithm rather than a per-flow TCP end-point algorithm and use a different curve.

VIII. Conclusions

We developed an AQM approach called DUK that operates based on a natural threshold and runtime measurements instead of relying on pre-specified or pre-tuned parameters. We found that DUK achieves similar performance in terms of link utilization but reduces queuing delays and flow completion times compared to two of its leading contemporaries.

Acknowledgements

We thank Professor Vishal Misra at Columbia University for many discussions related to this research. This material is based upon work supported by the National Science Foundation under Grant No. 1302688.

References

[1] J. Gettys and K. Nichols, "Bufferbloat: Dark buffers in the internet," *Communications of the ACM*, vol. 9, no. 11, pp. 57–65, 2011.
[2] K. Nichols and V. Jacobson, "Controlling queue delay," *Communications of the ACM*, vol. 55, no. 7, pp. 42–50, 2012.
[3] R. Pan *et al.*, "Pie: A lightweight control scheme to address the bufferbloat problem," ftp://ftpeng.cisco.com/pie/documents, 2013.
[4] G. White and D. Rice, "Active queue management algorithms for docsis 3.0: A simulation study of codel, sfq-codel and pie in docsis 3.0 networks," http://www.cablelabs.com, 2013.
[5] S. Floyd and V. Jacobson, "Random early detection gateways for congestion avoidance," *Networking, IEEE/ACM Trans. on*, vol. 1, no. 4, 1993.
[6] K. Ramakrishnan, S. Floyd, and D. Black, "The addition of explicit congestion notification (ECN)," RFC 3168.
[7] "The network simulator (ns-3)," http://www.isi.edu/nsnam/ns, 2014.
[8] H. Zhang *et al.*, "A self-tuning structure for adaptation in tcp/aqm networks," in *Globecom 2003*, vol. 22, no. 1, 2003, pp. 3641–3645.
[9] C. V. Hollot *et al.*, "On designing improved controllers for aqm routers supporting tcp flows," in *INFOCOM 2001*, vol. 3. IEEE, 2001.
[10] "Codel source code," http://git.kernel.org/cgit/linux/kernel/git/stable/linux-stable.git/tree/net/sched/sch_codel.c,.../include/net/codel.h, 2013.
[11] "Pie source code," ftp://ftpeng.cisco.com/pie, 2013.
[12] "Codel source code for ns-3," codereview.appspot.com/6463048, 2014.
[13] "Linux qdsic," https://wiki.archlinux.org, 2013.
[14] "Netem," http://www.linuxfoundation.org, 2013.
[15] M. Alizadeh *et al.*, "Dctcp: Efficient packet transport for the commoditized data center," *Proceedings of SIGCOMM*, 2010.
[16] C. Hollot *et al.*, "Analysis and design of controllers for aqm routers supporting tcp flows," *IEEE Trans on Automatic Control*, 2002.
[17] "Voice over ip-per call bandwidth consumption," www.cisco.com, 2015.
[18] R. Jain, D.-M. Chiu, and W. R. Hawe, *A quantitative measure of fairness and discrimination for resource allocation in shared computer system*. Eastern Research Laboratory, Digital Equipment Corporation, 1984.
[19] W.-c. Feng *et al.*, "Blue: A new class of active queue management algorithms," *Technical Report, UM CSE-TR-387-99*, 1999.
[20] C. Long *et al.*, "The yellow active queue management algorithm," *Computer Networks*, vol. 47, no. 4, pp. 525–550, 2005.
[21] B. Wydrowski and M. Zukerman, "Green: An active queue management algorithm for a self managed internet," in *ICC 2002*, vol. 4. IEEE, 2002.
[22] S. S. Kunniyur and R. Srikant, "An adaptive virtual queue (avq) algorithm for active queue management," *Networking, IEEE/ACM Transactions on*, vol. 12, no. 2, pp. 286–299, 2004.
[23] J. Hong *et al.*, "Active queue management algorithm considering queue and load states," *Computer comm*, vol. 30, no. 4, pp. 886–892, 2007.
[24] D.-M. Chiu and R. Jain, "Analysis of the increase and decrease algorithms for congestion avoidance in computer networks," *Computer Networks and ISDN systems*, vol. 17, no. 1, pp. 1–14, 1989.
[25] R. Jain, "A delay-based approach for congestion avoidance in interconnected heterogeneous computer networks," *ACM SIGCOMM Computer Communication Review*, vol. 19, no. 5, pp. 56–71, 1989.