Towards the Fairness of Traffic Policer

Danfeng Shan[1], Peng Zhang[1], Wanchun Jiang[2], Hao Li[1], Fengyuan Ren[3]
[1]*Xi'an Jiaotong University* [2]*Central South University* [3]*Tsinghua University*

IEEE INFOCOM 2021 - IEEE Conference on Computer Communications | 978-1-6654-0325-2/21/$31.00 ©2021 IEEE | DOI: 10.1109/INFOCOM42981.2021.9488761

***Abstract*—Traffic policing is widely used by ISPs to limit their customers' traffic rates. It has long been believed that a well-tuned traffic policer offers a satisfactory performance for TCP. However, we find this belief breaks with the emergence of new congestion control (CC) algorithms like BBR: flows using these new CC algorithms can easily occupy the majority of the bandwidth, starving traditional TCP flows. We confirm this problem with experiments and reveal its root cause as follows. Without buffer in traffic policers, congestion *only* causes packet losses, while new CC algorithms are *loss-resilient*, i.e. they adjust the sending rate based on other network feedback like delay. Thus, when being policed they will not reduce the sending rate until an unacceptable loss ratio for TCP is reached, resulting in low throughput for TCP. Simply adding buffer to the traffic policer improves fairness but incurs high latency. To this end, we propose FairPolicer, which can achieve fair bandwidth allocation without sacrificing latency. FairPolicer regards token as a basic unit of bandwidth and fairly allocates tokens to active flows in a round-robin manner. Testbed experiments show that FairPolicer can significantly improve the fairness and achieve much lower latency than other kinds of rate-limiters.**

***Index Terms*—Internet, Traffic Policing, Congestion Control, Packet Loss**

I. INTRODUCTION

On the Internet, an Internet Service Provider (ISP) usually needs to regulate its customers' traffic rate to enforce various network policies [1]–[5]. A common mechanism to enforce traffic rate is traffic policing, which is usually implemented by a bufferless token bucket [2], [6]. A token bucket policer does not contain buffer and just drops packets if the traffic rate is above the throttling rate. Thus, it is quite simple to be implemented in both software and hardware, enabling a network device to deploy hundreds of traffic policers. In addition, without queueing, the token bucket policer does not result in any latency inflation. Due to the above features, token bucket policer is widely adopted by ISPs [1]–[3].

The impact of traffic policing on network performance has been studied a lot in history [7]–[12]. It has been shown that TCP flows can achieve satisfactory performance as long as the parameters of traffic policing are well-tuned [10], [11]. However, this may no longer be true with the emergence of new congestion control (CC) algorithms recently [13]–[22]. These CC algorithms are quite different from the traditional CC algorithms used by TCP. In addition to packet loss, they use various network feedback to achieve both high throughput and low latency. Some of them [15], [16], [23], [24] have already been deployed in the production networks. Under this new trend, we find that *a serious throughput unfairness problem arises:* when flows using these CCs are competing with traditional TCP flows in a policer, the former can occupy the majority of the bandwidth, starving traditional TCP flows.

In this paper, we validate this problem through experiments and analyze its root causes (§III). We find that up to 99.8% of bandwidth can be occupied by flows using new CCs when contending with traditional TCP flows. Through analysis, we find the reason is that traffic policer can *only generate packet loss* as the signal of congestion, while the new and traditional CC algorithms have different sensitivity to packet losses. Traditional CC algorithms used by TCP are *loss-sensitive*: they reduce the sending rate once a packet loss is experienced. The new CC algorithms, on the other hand, are *loss-resilient*: in addition to packet loss, they adjust their sending rates based on other network feedback (e.g., delay). However, without a queue, these new CCs can not observe other congestion signals. Thus, the new CCs will not reduce their sending rates until a high packet loss ratio is reached, which is unacceptable for loss-sensitive TCP flows. As a result, the throughput of traditional TCP drops dramatically.

A straightforward approach is to add a queue into the traffic policer (which is called *traffic shaper*) so that the new CCs can obtain feedback about network delay when encountering congestion. However, this approach can incur a high queueing delay, which is unacceptable as many applications in today's Internet demand ultra-low latency [25], [26]. Another approach is to improve the CC algorithm at end hosts. However, it is hard for this approach to be adopted in real networks. This is because the service providers tend to deliver their data as fast as possible and may not be willing to concede bandwidth for the fairness purpose.

In this paper, we propose an active approach to tackle this problem: rather than passively relying on end hosts to friendly occupy the bandwidth, we propose to actively allocate bandwidth at the traffic policer. Specifically, we present *FairPolicer* (§IV), a low-latency traffic policer that can fairly allocate bandwidth among competing flows regardless of their CC algorithms. The observation behind FairPolicer is that token is the basic unit of bandwidth in token bucket policer. Thus, FairPolicer tries to fairly allocate tokens to active flows. To achieve this, FairPolicer divides the token bucket into multiple per-flow token buckets and allocates tokens to these buckets in a round-robin manner.

Although the basic idea is quite simple, to make it practical requires solving two key challenges. First, the number of active flows is dynamically changing. When a flow becomes inactive, the tokens in its bucket will never be used, leading to a waste of bandwidth. Second, with many concurrent flows, maintaining per-flow data requires lots of memory.

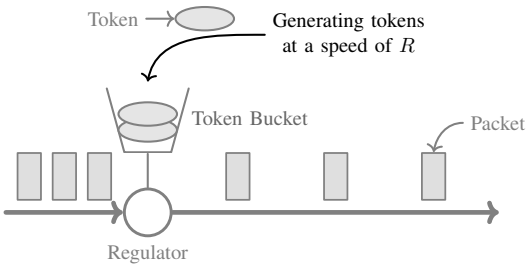


Fig. 1: Token Bucket Policer

We use two methods to address the above challenges and make FairPolicer practical. First, rather than distributing tokens among the per-flow token buckets, FairPolicer puts all available tokens in a centralized global token bucket and maintains the residual bucket space of each flow instead. Allocating a token to a flow is achieved by decreasing its residual bucket space. In this way, when a flow becomes inactive, the tokens allocated to this flow are in the global bucket. We can simply delete the bucket of the flow without wasting tokens. Second, FairPolicer leverages a Count-Min Sketch structure [27] to store the per-flow token occupancy, which enables FairPolicer to maintain per-flow data with a small memory footprint. As shown in §V, FairPolicer can scale to 1,000 flows with only 32KB memory.

We implement a prototype of FairPolicer in the Linux kernel[1] and evaluate it on a real testbed (§V). Our experiments demonstrate that (1) FairPolicer can ensure fair bandwidth allocation among flows with different CCs, and (2) FairPolicer can achieve low latency for short flows. Specifically, when loading a web page, FairPolicer can reduce the tail latency by up to 14.0× and 15.1× compared to token bucket policer and traffic shaper, respectively.

In sum, our contribution is three-fold:

- To the best of our knowledge, we are the first to discover and validate the unfairness problem with new and traditional CC algorithms competing in a traffic policer.
- We propose FairPolicer, a new traffic policer that can fairly allocate bandwidth among contending flows regardless of their CC algorithms.
- We prototype FairPolicer and evaluate it on a real testbed to demonstrate that it can greatly improve the fairness while still achieving a much lower latency compared with other kinds of rate-limiters.

II. BACKGROUND

In this section, we briefly describe how a traffic policer works and discuss the trend of CC algorithms on the Internet.

A. Traffic Policing on the Internet

Traffic policing is widely used by ISPs to enforce a specific traffic rate. For example, T-Mobile's "Binge On" service provides zero-rated access to video streaming while limiting the traffic rate to 1.5Mbps with a traffic policer [1]. Some ISPs use traffic policing to enforce their customers' traffic rates below the bandwidth of their data plans [2], [3].

Traffic policing is mostly implemented by the token bucket algorithm [2], [6], which works as follows. As shown in Fig. 1, to throttle the traffic rate to R, the policer generates tokens at a rate of R and puts these tokens into a bucket. When a packet arrives at the policer, a regulator checks whether there are enough tokens in the bucket. If the number of tokens is no less than the packet length (denoted by l), the policer delivers the packet and removes l tokens from the bucket. Otherwise, the packet is dropped.

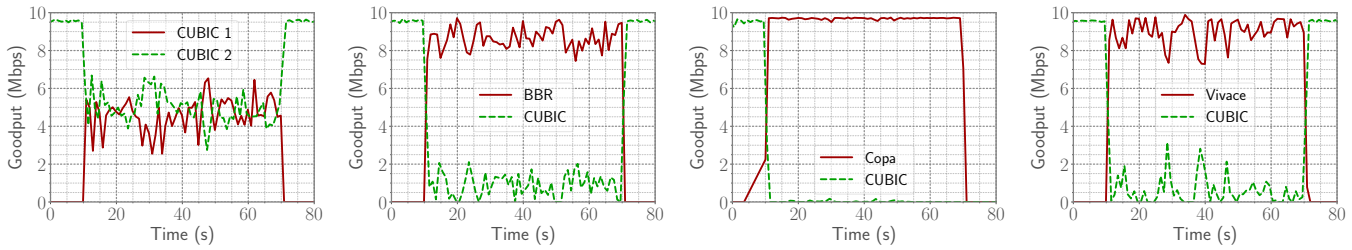
Compared to other kinds of rate-limiters, traffic policer has two advantages. First, it is very easy to be implemented. It only requires a counter and a timer to implement its core algorithm. The counter records the number of tokens in the bucket, and the timer increases the counter at a rate of R. With such a simplicity, one can easily deploy a large number of traffic policers inside a single network device with little cost. Second, without queueing, the traffic policer does not inflate network latency. This is of great importance as the bandwidth of today's Internet has increased a lot and latency becomes the main concentration for many network applications [26].

B. Congestion Control Algorithms

Congestion Control (CC) is crucial to the user experience of Internet applications. Currently, CUBIC [28] — the default CC in Linux and Windows [29] — is perhaps the most popular CC algorithm on the Internet. However, as the network becomes complex, its performance is far from satisfactory. On the one hand, as a loss-based CC, CUBIC tends to fill the bottleneck buffer regardless of the buffer size, causing the "bufferbloat" problem that flows experience very high latency. On the other hand, CUBIC cannot distinguish between congestion-induced packet loss and non-congestion-induced packet loss (e.g., random packet loss), leading to a degradation of throughput in some lossy scenarios.

Due to the above reasons, lots of new CC algorithms have been proposed recently [13]–[22]. In addition to packet loss, these CC algorithms also incorporate other network feedback to achieve low latency and high throughput simultaneously. For example, BBR [16] maintains an estimate of the round-trip propagation time (i.e., round-trip time without queueing) and the bottleneck bandwidth. Utilizing these kinds of feedback, it sends packets at a rate of bottleneck bandwidth to achieve the maximum throughput and limits the inflight packets to $O(\text{BDP})$ to achieve low queueing delay, where BDP is the product of round-trip propagation time and bottleneck bandwidth. BBR has already replaced CUBIC on Google's B4 and been deployed on YouTube video servers since 2016 [16]. Copa [18] is a delay-based CC algorithm. It adjusts its congestion window towards a target value, which is inversely proportional to the estimated queueing delay. Copa has been deployed in Facebook for live video upload [23]. PCC Allegro [14] and PCC Vivace [17] regard the network as a black box and adjust the sending rate to maximize a utility value, which

[1]Source code available at https://github.com/ants-xjtu/fairpolicer.



(a) 2 CUBICs (b) 1 CUBIC vs. 1 BBR (c) 1 CUBIC vs. 1 Copa (d) 1 CUBIC vs. 1 PCC Vivace

Fig. 2: Goodputs of CUBIC flows and BBR/Copa/PCC Vivace flows when rate-limited by the same token bucket policer

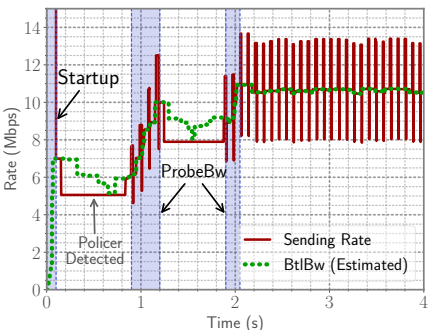


Fig. 3: Temporal behavior of BBR

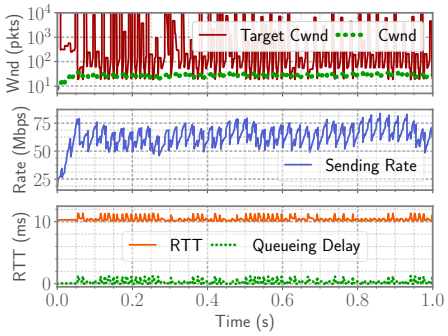


Fig. 4: Temporal behavior of Copa

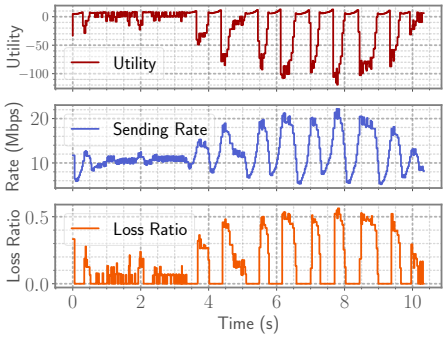


Fig. 5: Temporal behavior of PCC Vivace

is a function of the packet loss ratio, RTT (or its gradient), and sending rate.

III. The Unfairness of Traffic Policer

Although these recently proposed CC algorithms are designed with TCP-Friendliness in mind, they have not considered the wide presence of traffic policers. As a result, flows using these new CCs can occupy the majority of bandwidth when competing with traditional TCP flows. In the following, we use real experiments to demonstrate this problem and analyze its root cause.

A. Experimental Results

We have built a testbed with four hosts connected to a server-emulated switch (more details in §V-A). One host serves as a receiver and other hosts serve as senders. The traffic rate is throttled to 10Mbps by a 50KB token bucket. The base round-trip time (RTT) between hosts is 10ms.

As shown in Fig. 2a, two CUBIC flows can fairly share the bandwidth in the traffic policer. However, with one CUBIC flow and one BBR flow (Fig. 2b), the BBR flow can occupy the majority of bandwidth, starving the CUBIC flow. The same phenomenon can be observed with Copa and PCC (Fig. 2c and Fig. 2d). Specifically, competing with a CUBIC flow, the BBR, Copa, or PCC Vivace flow can occupy 90.6%, 99.8%, and 93.8% of bandwidth, respectively.

B. Analysis of the Unfairness Problem

1) BBR: Fig. 3 shows the dynamic behavior of BBR when competing with a CUBIC flow in a traffic policer. The solid line depicts the sending rate and the dashed line depicts the estimated bottleneck bandwidth. We can observe that the BBR flow occupies the majority of the bandwidth by two steps.

First, when the BBR flow starts, it enters into a `Startup` state. Like the slow start phase of TCP, the BBR flow in this state doubles its sending rate for each round. Unlike the slow start phase, BBR does not reduce the sending rate when encountering packet losses. It stays in this state until it does not observe an increment of delivery rate. On the other hand, the CUBIC flow will continuously reduce its sending rate when encountering packet losses. When competing with the CUBIC flow, the BBR flow will quickly occupy the bandwidth freed by the CUBIC flow. As a result, after the `Startup` state, BBR can occupy most of the bandwidth. As shown in Fig. 3, the BBR flow increases its sending rate to 7Mbps at the end of the `Startup` state.

Second, even if the BBR flow has not occupied the majority of the bandwidth during the `Startup` state, it can gradually approach the throttling rate by increasing its sending rate during the `ProbeBW` state. Specifically, in the `ProbeBW` state, BBR periodically probes the available bandwidth through increasing its sending rate by 25% for 1 RTT. During this time, the CUBIC flow will experience more packet losses and concede some bandwidth, which is quickly occupied by the BBR flow. Consequently, after probing, the BBR flow can observe a higher value of bottleneck bandwidth. As shown in Fig. 3, during [0.9s, 1.2s] and [1.9s, 2.1s], BBR increases its average sending rate to 10Mbps and 10.5Mbps, respectively[2].

Although BBR contains a policer detector, it is unable to discover the existence of the policer most of the time. As

[2]As one might have noticed, BBR can slightly overestimate the bottleneck bandwidth. This is because traffic policer allows some degree of bursty transmissions. Consequently, the delivery rate can be temporarily larger than the policing rate.

shown in Fig. 3, BBR is only able to detect the policer in [0.15s, 0.82s] and [1.24s, 1.87s]. Besides, even after detecting the policer, a BBR flow will send data at its estimated throttling rate, which can be larger than the fair share rate. For example, during [1.24s, 1.87s] in Fig. 3, the average sending rate is ∼8Mbps, which is 60% higher than the fair share rate.

2) Copa: Fig. 4 depicts the dynamic behavior of Copa when competing with 1 CUBIC flow in a token bucket policer. Copa estimates the queueing delay by observing the current RTT and global minimum RTT. It sets a target congestion window according to the estimated queueing delay. As a traffic policer does not contain a queue, the estimated queueing delay is very small. As shown in Fig. 4, the queueing delay estimated by Copa is smaller than 0.1ms most of the time. As a result, the target congestion window is several orders of magnitude larger than the Bandwidth-Delay Product (BDP). And the sending rate of Copa is much higher than the throttling rate. As shown in Fig. 4, the congestion window is around 30 packets and the sending rate is around 60Mbps. With such a high sending rate, the Copa flow occupies the majority of the bandwidth, starving the CUBIC flow.

3) PCC Vivace: PCC Vivace adjusts its sending rate according to a utility function. Its default utility function is

$$u = x^t - bx_i\frac{\mathrm{d}(RTT)}{\mathrm{d}T} - cxL \tag{1}$$

where t, b, c are constants, x is the sending rate, $\frac{\mathrm{d}(RTT)}{\mathrm{d}T}$ is the RTT gradient, and L is the loss ratio. When rate-limited by a token bucket policer, the RTT does not change much, namely, $\frac{\mathrm{d}(RTT)}{\mathrm{d}T} \approx 0$. Consequently, Eq. (1) can be rewritten as

$$u \approx x^t - cxL \tag{2}$$

According to [17] and [30], Eq. (2) does not decrease until the packet loss ratio achieves 6.3%. However, such a packet loss ratio is too high for CUBIC. The throughput of a CUBIC flow drops to 1% at a loss rate of 6% [17]. As a result, the majority of the bandwidth is occupied by the PCC Vivace flow.

Fig. 5 depicts the dynamic behavior of PCC Vivace when competing with 1 CUBIC flow in a token bucket policer. The sending rate of the PCC Vivace flow is over the throttling rate most of the time.

C. Summary

In summary, the causes of the throughput unfairness problem are two-fold. On the one hand, the traffic policer *only generates packet loss* as the network feedback about congestion status. On the other hand, CUBIC and new CC algorithms have quite different tolerability to packet loss. CUBIC is *loss-sensitive*; it reduces its sending rate dramatically when encountering a high packet loss ratio. The new CC algorithms are *loss-resilient*; they also rely on other network feedback to determine the sending rate. However, in the policer, packet loss is the *only network feedback* about congestion status. Without the help of other network feedback, these loss-resilient CC algorithms do not reduce their sending rate until the loss ratio reaches an unbearable value for CUBIC. As a result, the majority of bandwidth is occupied by these loss-resilient CC flows, starving the loss-sensitive CUBIC flows.

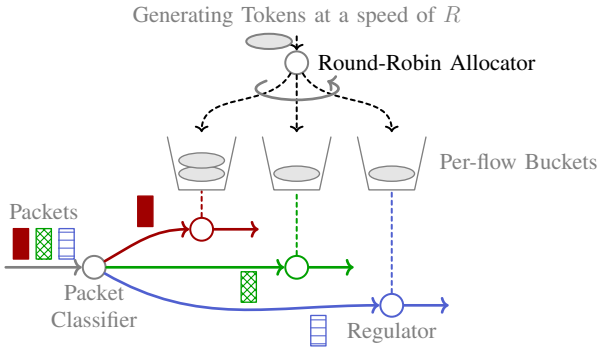


Fig. 6: Basic Idea

IV. Mitigating the Unfairness Problem

In this section, we first discuss the limitations of another rate-limiting method — traffic shaping. Then we propose FairPolicer to fairly allocate bandwidth regardless of the CC algorithms of contending flows.

A. Limitations of Traffic Shaping

Traffic shaping is another approach to achieve rate-limiting. Different from the traffic policer, a traffic shaper contains a queue and buffers the traffic above the throttling rate [6]. With a traffic shaper, senders can be aware of latency inflation when the network becomes congestion. The loss-resilient CCs will reduce their sending rates before overwhelming the buffer, protecting loss-sensitive flows from starving for bandwidth.

However, traffic shaping has three drawbacks. First, loss-based CCs tend to fill the buffer of the shaper, resulting in high in-network latency. This is quite unacceptable as latency is the limiting factor for many Internet applications today [26], such as online gaming, financial trading, and VoIP. Second, the traffic shaper cannot guarantee fair bandwidth allocation in all cases. For example, if the packet buffer is not large enough, BBR can still occupy the majority of bandwidth [31], [32]. Third, as the traffic shaper needs some memory to store packets, it can dramatically increase the cost of a rate-limiter, especially when hundreds of rate-limiters need to be deployed in a single network device.

B. Basic Idea

Given the limitations of traffic shaping as discussed above, we set out to design a new traffic policer to solve the unfairness problem.

We observe that a cause of the unfairness problem is a lack of bandwidth allocation mechanism in the current traffic policing scheme. Specifically, tokens are given to whatever packets that arrive, without enforcing a fair share of tokens among flows. Based on this observation, we propose FairPolicer. The basic idea of FairPolicer is splitting the token bucket to multiple per-flow token buckets and fairly allocating tokens to each flow. Specifically, as shown in Fig. 6, On the data path (solid line), FairPolicer classifies the arriving packets

based on their flows. Different kinds of packets are regulated by different token buckets. On the control path (dashed line), FairPolicer generates tokens at a speed of throttling rate (denoted by R). The generated tokens are allocated to non-full token buckets in a round-robin manner.

C. Challenges

However, although the basic idea seems to be simple, its realization faces two practical challenges.

First, the number of active flows is varying. After a flow finishes its transmission, its bucket will be full of tokens. These tokens will never be used and thus are wasted. One solution is to reallocate these tokens to other active flows. However, it is not easy to implement this solution in the hardware because it requires an arbitrary number of hardware operations. On the other hand, when a flow becomes active, its bucket is empty at the beginning. The bucket should be filled with tokens immediately. Otherwise, the arriving packets will be dropped, degrading the performance of the flow. The tokens to fill the bucket should be moved from other buckets rather than generated all at once. Otherwise, FairPolicer is not able to limit the overall traffic rate to its throttling rate. However, moving tokens from other non-empty buckets requires lots of non-trivial operations in hardware. What's worse, lots of flows can become active and inactive in a very short time, resulting in difficulty in hardware implementation.

Second, the number of concurrent flows could be very large on the Internet [33], [34]. Accurately maintaining per-flow state in the policer requires a lot of memory, which burdens the network devices deploying the policers.

We use two methods to address the above challenges and make FairPolicer practical. First, rather than distributing the available tokens among the per-flow token buckets, we put them into a centralized global bucket. In each per-flow bucket, we maintain the *residual bucket space* instead of available tokens. The residual bucket space denotes the number of used tokens of the flow. Thus, we use *the number of occupied tokens* to denote the residual bucket space. Allocating a token to a flow is achieved by decreasing the number of its occupied tokens. In this way, FairPolicer can be adaptive to the variation of active flows. Specifically, when a flow becomes inactive, the number of its occupied tokens becomes zero. Reallocating its tokens can be simply achieved by raising the bucket capacity of other flows. When a flow becomes active, its number of occupied tokens is zero at the beginning, which denotes that its bucket is full of tokens. No further operations are needed to fill the bucket except that FairPolicer needs to reduce the bucket capacity of other flows to limit the overall number of occupied tokens.

Second, we use the Count-Min Sketch structure [27] to estimate the number of occupied tokens for each flow. A Count-Min Sketch is a two-dimension array of counters with d rows and w columns. Each row is associated with a separate hash function. The d hash functions should be pairwise independent. A Count-Min Sketch supports two operations. (1) `Update(`f`,`c`)` updates the counter of flow f by c, where

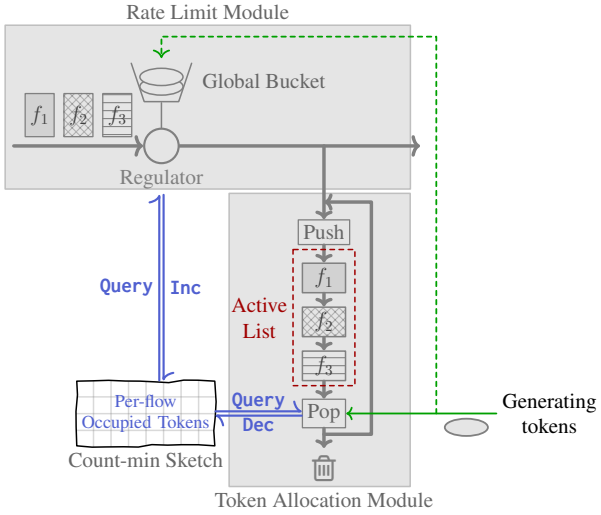


Fig. 7: Structure of FairPolicer

c is a given integer, which can be either positive or negative. To achieve this operation, the Count-Min Sketch uses d hash functions to locate a counter in each row and updates all located counters by c. (2) `Query(f)` queries the counter of flow f. To achieve this operation, the Count-Min Sketch uses d hash functions to locate d counters and returns the minimum counter. Both operations can be easily implemented in the hardware [35]. Some recent studies have shown that Count-Min Sketch can track per-flow statistics in sub-linear space [35], [36]. In §IV-E and §V-D, we'll further show that Count-Min Sketch can achieve a high estimation accuracy with a small memory footprint.

D. Details of FairPolicer

Fig. 7 shows the structure of FairPolicer, which consists of two modules: *rate limit module* and *token allocation module*.

The rate limit module restricts the traffic rate of each flow. FairPolicer preserves the key mechanism of token bucket policer to limit the overall traffic rate. Specifically, FairPolicer generates tokens at a rate of R and puts them into a global bucket; an arriving packet is allowed to pass the regulator only if there are enough tokens in the global bucket.

Furthermore, FairPolicer limits the per-flow traffic rate through restricting the number of per-flow occupied tokens by a threshold. The threshold represents the bucket capacity of each flow. To ensure fairness, all flows use the same threshold. Specifically, an arriving packet of flow i is allowed to pass the regulator if and only if

$$o^{(i)} < T \tag{3}$$

where $o^{(i)}$ is the number of occupied tokens for flow i and T is the threshold. As is mentioned before, the threshold is adjusted according to the number of active flows. With n active flows, FairPolicer sets the threshold as

$$T = \frac{B}{n+1} \tag{4}$$

where B is the capacity of the global token bucket. We set the threshold in this way for two reasons. First, FairPolicer tries

to reserve $B/(n+1)$ free tokens so that newly arriving flows won't starve for available tokens. Second, except the reserved ones, the remaining tokens can be equally distributed among the active flows.

One may concern that setting threshold as Eq. (4) may lead to a waste of bandwidth, especially when n is very small and lots of tokens are reserved. We argue that the bandwidth waste is negligible in the long run. Consider the case when $n = 1$, half of the tokens are reserved. For a period of $[0, t]$, FairPolicer generates $R \cdot t$ tokens in total. During this period, $B/2$ tokens are always reserved. Consequently, the fraction of wasted tokens is given by $B/(2Rt)$, which becomes very small with a large t.

Nevertheless, such a threshold calculation raises two implementation issues. First, FairPolicer needs to determine the number of active flows. Second, calculating the threshold needs a divider, which can introduce some implementation overhead.

To simplify the implementation, we borrow the idea of DT [37] to approximately adjust the threshold as in (4). Specifically, we set the threshold as

$$T = k = B - \sum_{i=1}^{n} o^{(i)} \tag{5}$$

where k is the number of available tokens in the global bucket. With Eq. (5), FairPolicer dynamically adjusts T according to the current unused tokens. As all flows share the same threshold, in the steady state, FairPolicer can fairly allocate tokens to all bandwidth-demanding flows.

To show how Eq. (5) can approximate Eq. (4), consider the case that there are n active flows whose traffic rates are all higher than R/n. In the beginning, some flows arrived earlier and may have occupied much more tokens than others. As new flows arrive and occupy the free tokens, the threshold T becomes smaller, obliging previously arrived flows to concede tokens to newly arrived flows. Eventually, in the steady state, all flows will occupy the same number of tokens, namely, $o^{(1)} = o^{(2)} = \cdots = o^{(n)} = T$. According to (5), the threshold can be given by

$$T = B - \sum_{i=1}^{n} o^{(i)} = B - nT \tag{6}$$

which is equivalent to (4).

Finally, when a packet is allowed to pass the regulator, FairPolicer increases the number of occupied tokens by updating the Count-Min Sketch.

The token allocation module fairly allocates the generated tokens to the active flows. The core of this module is maintaining the active flow list. Ideally, to determine whether a flow is active, we need to observe its traffic during a time window. However, such a method complicates the implementation of FairPolicer, especially in hardware.

Fortunately, we observe that there is no need to maintain a "real" active flow list. Specifically, FairPolicer does not need to allocate tokens to the flows whose buckets have already been filled with tokens. In other words, FairPolicer only needs to maintain a list of flows whose numbers of occupied tokens are non-zero.

To maintain such a list, FairPolicer adds a flow into the list whenever its number of occupied tokens becomes non-zero, and removes a flow from the list whenever its number of occupied tokens becomes zero. The former operation is conducted when a packet is allowed to pass the regulator. Specifically, FairPolicer pushes the corresponding flow into the list if the token occupancy before the packet arrival is zero. The latter operation is conducted after a token is generated. Whenever generating a token, FairPolicer pops a flow from the list and allocates the token to the flow by decreasing its number of occupied tokens. Then FairPolicer determines whether this flow still has token occupancy by querying the Count-Min Sketch. If the number of occupied tokens for this flow becomes zero, FairPolicer removes this flow from the list. Otherwise, FairPolicer appends the flow to the end of the list.

However, with a high throttling rate (e.g., 10Gbps), the operation frequency can be very high. To reduce the implementation complexity in such environments, we can generate a number of tokens, say, 10KB at a time and allocate them to a flow. Such method trades off bandwidth. Specifically, if a flow occupies less than 10KB tokens, some generated tokens are wasted. However, we think that this is not a serious problem in high-bandwidth environments.

E. Accuracy of Count-Min Sketch

Count-Min Sketch can over-estimate a flow's occupied tokens in case of hash collisions, causing flows to obtain bandwidth smaller than their fair share rate. Nevertheless, FairPolicer can achieve a high estimation accuracy with small memory for the following two reasons.

First, the estimation error is bounded. The accuracy of a Count-Min Sketch is highly related to the total number of increments to its counters. In FairPolicer, the total number of increments is no larger than B. Therefore, the estimation error is bounded. Formally, we have the following theorem, which can be easily derived from Theorem 1 in [27].

Theorem 1. *The estimation error is within ϵB with a probability of $1 - \delta$, where $\epsilon = e/w$ and $\delta = 1/e^d$.*

Second, a single counter only needs a small amount of memory. For a single counter, its maximum value is no larger than B. Therefore, it is sufficient to use $\Theta(\log_2 B)$ bits for a counter. For example, if a counter is at a granularity of 40B and $B = 100$KB, then 2B memory is sufficient for a single counter. A 4×1024 Count-Min Sketch only needs 8KB memory. Therefore, it is possible to employ a large Count-Min Sketch in FairPolicer.

F. Parameter Settings

FairPolicer has two kinds of parameters: bucket capacity B and Count-Min Sketch size (i.e., depth d and width w). The Count-Min Sketch size can be set according to Theorem 1. In this part, we mainly analyze the setting of bucket capacity B.

TABLE I: Key Notations

Not.	Description
B	Bucket capacity / Maximum burst size
R	Throttling rate
T	Token threshold
t	Time elapsed since the last window reduction
$w(t)$	Congestion window at time t
$w_{\max}$	The window size just before the last reduction
C	The scaling factor of CUBIC
β	The window reduction factor of CUBIC
D	Round-trip time
n	Number of active flows

We focus on the influence of bucket capacity on the loss-sensitive flows. This is because loss-resilient flows usually send packets at a rate higher than the fair share rate regardless of how large the bucket capacity is. For loss-sensitive flows, there are two considerations when setting the bucket capacity. On the one hand, the bucket capacity should be large enough so that the throughput of a flow can achieve its fair share rate [11]. On the other hand, the bucket capacity should not be too large. This is because a larger bucket capacity allows a higher degree of traffic burstiness, which can result in a higher probability of packet dropping in the successive hops [2]. In this part, we aim to derive the smallest bucket capacity to ensure full throughput of a flow. The key notations are summarized in TABLE I for the sake of terseness.

We start from a simple scenario that only one flow is passing through a token bucket policer, which has a bucket capacity of B. We consider CUBIC as it is the most common loss-sensitive CC algorithm. The congestion window (denoted by $w(t)$) of CUBIC is determined by the following function [28]:

$$w(t) = C(t-K)^3 + w_{\max} \tag{7}$$

where C is a scaling factor, t is the time elapsed from the last window reduction, $w_{\max}$ is the window size just before the last reduction, and $K = \sqrt[3]{w_{\max}\beta/C}$. The parameter β is the window reduction factor. The average sending rate of the CUBIC flow can be given by $w(t)/D$, where D is the round-trip time.

Fig. 8 shows the steady-state evolution of the congestion window for a CUBIC flow, which can be divided into two phases. In Phase 1 (i.e., $[0, \tau)$), the sending rate of the flow is lower than the throttling rate. Thus, the policer is accumulating tokens in the bucket. In Phase 2 (i.e., $(\tau, K]$), the sending rate of the flow is higher than the throttling rate. Thus, the flow is consuming tokens in the bucket until the bucket is empty.

To ensure that a flow can achieve a rate of R, all generated tokens during $[0, K]$ should be consumed by the flow. In other words, the bucket should not be overflowed in Phase 1. Formally, we should have

$$R\tau - \int_0^{\tau} \left[\frac{C(t-K)^3 + w_{\max}}{D}\right] \mathrm{d}t \leqslant B \tag{8}$$

To solve the above inequality, we need to determine $w_{\max}$ and τ. As the bucket is empty at time K, we have

$$\int_0^{K} \left[\frac{C(t-K)^3 + w_{\max}}{D}\right] \mathrm{d}t = R \cdot K \tag{9}$$

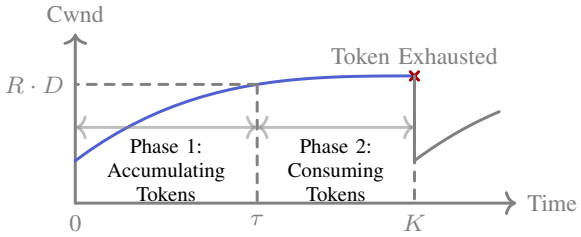


Fig. 8: Window evolution of CUBIC in the steady state

Solving for $w_{\max}$ from (9), we have

$$w_{\max} = \frac{4}{4-\beta} RD \tag{10}$$

On the other hand, at time $t = \tau$, the sending rate of CUBIC flow is equal to the throttling rate, namely, $w(\tau) = RD$. Solving for τ from it, we have

$$\tau = \left[\sqrt[3]{\frac{4\beta}{C(4-\beta)}} - \sqrt[3]{\frac{\beta}{C(4-\beta)}}\right] \sqrt[3]{RD} \tag{11}$$

Plugging (10) and (11) into (8), we get

$$B \geqslant \frac{G}{D} \cdot (RD)^{\frac{4}{3}} \tag{12}$$

where

$$G = \frac{3}{4\sqrt[3]{C}} \left(\frac{\beta}{4-\beta}\right)^{\frac{4}{3}} \tag{13}$$

Now we consider a more complicated scenario where n active flows are competing for bandwidth in FairPolicer. In this case, we should replace R with R/n and B with $B/(n+1)$ in inequality (12), which yields

$$B \geqslant \left(\frac{1}{n^{\frac{1}{3}}} + \frac{1}{n^{\frac{4}{3}}}\right) \frac{G}{D} (RD)^{\frac{4}{3}} \tag{14}$$

Note that the right part of (14) reaches its maximum at $n = 1$. Thus, we have

$$B \geqslant \frac{2G}{D} (RD)^{\frac{4}{3}} \tag{15}$$

According to the RFC [38], the parameters of CUBIC should be set as $\beta = 0.3$ and $C = 0.4$. Thus, we have $G = 0.036$. Substituting it into (15) yields $B \geqslant \frac{0.072}{D}(RD)^{\frac{4}{3}}$.

On the other hand, the bucket capacity should be as small as possible to reduce the traffic burstiness. Thus, we should set the bucket capacity as

$$B = \frac{0.072}{D} (RD)^{\frac{4}{3}} \tag{16}$$

Finally, to determine B in above equation, we need to set D (i.e., RTT). There are two considerations. First, obtaining RTT on the fly is not easy; we need to measure RTT and set the parameter offline. Second, various flows tend to have different RTTs. To ensure that all flows can achieve the fair share rate, we should set D as the maximum RTT of all flows in the network. For networks whose RTT distribution is extremely long-tailed, we can choose a RTT (e.g., 99th percentile) such that the majority flows can achieve fair share rate while the bucket size is not too high.

V. Evaluation

In this section, we use testbed experiments and ns-3 simulations to answer the following three key questions:

- **Can FairPolicer ensure fair bandwidth allocation?** In §V-B, we show that FairPolicer can fairly allocate bandwidth to contending flows with various CCs, including CUBIC, BBR, Copa, PCC Vivace, and aggressive UDP.
- **Does FairPolicer achieve low latency for short flows?** In §V-C, we show that FairPolicer can reduce the average web page load latency by up to 5.1× and the tail web page load latency by up to 15.1×.
- **Can FairPolicer scale to a large number of flows?** In §V-D, we show that FairPolicer can achieve a misestimation fraction of 0.001 using only 32KB memory with 1K-2K concurrent flows.

A. Implementation and Testbed Setup

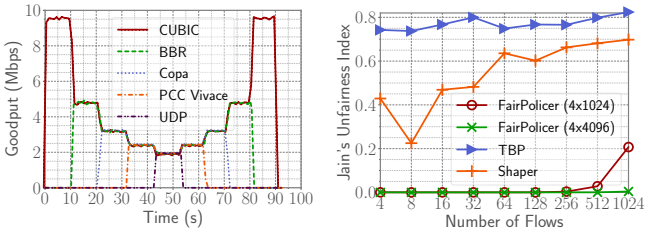
We implement a prototype of FairPolicer as a new queueing discipline (`qdisc`) in the Linux kernel, which lies between the IP layer and the network interface driver layer. A `qdisc` contains a packet queue and uses an enqueue callback and a dequeue callback to perform the enqueue and dequeue operations, respectively. We implement all FairPolicer functions in the enqueue function and manually set the maximum queue size to one packet to emulate a bufferless policer.

We build a testbed with 4 servers connected by another server emulating a switch. Each server is a Dell PowerEdge R730 server with a 16-core Intel Xeon E5-2620 2.10GHz CPU, 16GB memory, a 1TB hard disk, and two Broadcom NetXtreme BCM5720 Gigabit Ethernet NICs. The server-emulated switch has four Intel 82580 Gigabit Ethernet NICs connecting to four servers. All servers run Ubuntu 18.04 with Linux kernel 4.15.0. We use `tc-netem` to enlarge the RTT to 10ms.

In the server-emulated switch, we consider three kinds of rate-limiters: FairPolicer, token bucket policer (TBP), and traffic shaper. These rate-limiters are configured on the network interface connecting to the receiver. For FairPolicer, the Count-Min Sketch has a default size of 4×1024 counters. The bucket size is set to 180KB according to Eq. (16)[3]. We use `tc-tbf` to emulate TBP and traffic shaper. For TBP, we set the queue size to 1600B to emulate a bufferless TBP. We set the bucket size to 90KB according to Eq. (12). For traffic shaper, we set both the bucket size and the queue size to 50KB. The throttling rate of all rate-limiters is 10Mbps.

In the end hosts, we consider two kinds of CCs: loss-sensitive CC and loss-resilient CC. For loss-sensitive CC, we consider CUBIC as it is the default CC in mainstream operating systems and thus is typical. For loss-resilient CC, we consider BBR, Copa, and PCC Vivace. CUBIC and BBR are officially supported by Linux 4.15.0. We use `iperf3` to generate CUBIC and BBR traffic. We use the user-space implementation of Copa [39] and PCC Vivace [40] to generate their traffic. CUBIC traffic and other traffic are generated in separate senders so as not to affect each other before arriving at the rate-limiter.

[3]Note that in Eq. (16) the unit of D is seconds and the unit of R is MSS (Maximum Segment Size) per second.



(a) Goodput Evolution (b) Unfairness Index

Fig. 9: Fairness among loss-sensitive flows and loss-resilient flows

B. Fairness

Temporal Fairness: First, we evaluate the fairness of bandwidth allocation among CUBIC, BBR, Copa, PCC Vivace, and an aggressive UDP flow. The UDP flow sends traffic at line rate (i.e., nearly 1Gbps) at the sender and is completely unaware of packet loss. In this experiment, we start a flow every 10 seconds for the first 50 seconds and terminate a flow every 10 seconds for the next 50 seconds. Fig. 9a shows the goodput of each flow. FairPolicer can guarantee a fair allocation of bandwidth among different kinds of CCs.

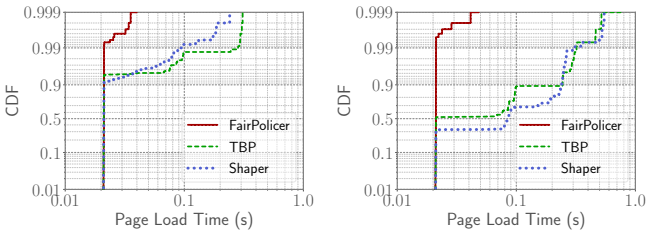
Fairness Index: Next, we use Jain's fairness index [41] to quantitatively evaluate the goodput fairness of FairPolicer, TBP, and traffic shaper. Jain's fairness index is given by $(\sum_i x_i)^2/(n \cdot \sum_i x_i^2)$, where x_i denotes the throughput of the i-th flow. The fairness index ranges from $1/n$ (worst case) to 1 (best case), where n is the number of flows.

In this experiment, we start the same number of CUBIC, BBR, Copa, and PCC Vivace flows simultaneously. Fig. 9b shows the Jain's unfairness index (1 - Jain's fairness index) with different kinds of rate-limiters. FairPolicer can achieve significantly better fairness than TBP and traffic shaper. Specifically, the unfairness index is within 0.004 when the number of concurrent flows is within the sketch size. In comparison, the unfairness indexes of TBP and traffic shaper are 0.74-0.82 and 0.48-0.70, respectively.

When the number of concurrent flows exceeds the sketch size, the Count-Min Sketch may overestimate the number of occupied tokens, resulting in goodput unfairness. Thus, the sketch size of FairPolicer should be appropriately configured. Nevertheless, as is analyzed, the Count-Min Sketch of FairPolicer does not consume much memory. Specifically, with only 32KB memory ($4 \times 4096 \times 2$B), FairPolicer can achieve fair bandwidth allocation among 1,000 flows.

C. Latency

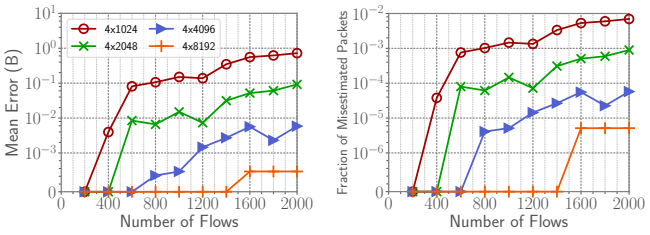
We now demonstrate how FairPolicer achieves low latency for short data transmission. In this experiment, we generate two kinds of traffic: HTTP traffic and background traffic. Both kinds of traffic use CUBIC as their CC algorithm. We set up an NGINX web server at a sender and let a receiver periodically



(a) Light-load background traffic (Pareto scale = 0.1)

(b) Heavy-load background traffic (Pareto scale = 0.01)

Fig. 10: Distributions of web page load time



(a) Mean error

(b) Fraction of misestimation

Fig. 11: Accuracy of Count-Min Sketch

requests a 10KB web page. The requests follow a Poisson process with an average rate of 1 request per second. We use D-ITG [42] to generate the background traffic. The inter-arrival time of packets follows a Pareto distribution with shape parameter 0.9 [43]. We vary the scale parameter to change the traffic load. The experiment lasts for 1,000 web requests.

Fig. 10 shows the CDF of page load time with different kinds of rate-limiters. Among the three rate-limiters, FairPolicer can achieve the shortest page load time. Specifically, with light-load background traffic (Fig. 10a), over 90% of the web requests can be finished within 22ms for all kinds of rate-limiters. However, TBP and traffic shaper has a much longer tail page load time than FairPolicer. Specifically, for traffic shaper, the 99th-percentile page load time is 92.9ms, which is over 4.4× longer than that of FairPolicer. This is because the background traffic can cause queue build-up in the traffic shaper and web requests may experience some queueing latency. In comparison, FairPolicer does not contain a queue and thus can achieve low latency. For TBP, the 99th percentile is 281ms, which is 13.2× longer than that of FairPolicer. This is because the background traffic can exhaust the tokens and TBP may drop the packets of web requests. As a result, the web requests may experience retransmission timeouts, significantly extending the page load latency. In comparison, FairPolicer does not allow the background traffic to use up all tokens, effectively avoiding packet droppings of web request flows.

With heavy-load background traffic (Fig. 10b), FairPolicer achieves much shorter page load time than traffic shaper and TBP. Specifically, FairPolicer can achieve 5.1× and 3.4× shorter average page load time than traffic shaper and TBP, respectively. And FairPolicer can achieve 14.0× and 15.1× shorter 99th-percentile page load time than TBP and traffic shaper, respectively. This is because traffic shaper and TBP can drop some packets of web requests due to the lack of buffer space and tokens, respectively, resulting in packet retransmissions and retransmission timeouts. In comparison, FairPolicer can isolate the web request traffic and background traffic, effectively avoiding packet droppings of web request flows.

D. Accuracy of Count-Min Sketch

In this part, we use ns-3 simulations [44] to evaluate the accuracy of the Count-Min Sketch with a large number of concurrent flows. We use a dumbbell topology with 100 senders and 100 receivers. The traffic rate of the bottleneck link is throttled to 80Mbps. The traffic is generated based on a Poisson process. The overall packet arrival rate is equal to the throttling rate.

Fig. 11a shows the mean estimation error of occupied tokens with different sketch sizes. FairPolicer can achieve high estimation accuracy with a small memory footprint. Specifically, the mean error is within 0.01B with a sketch size of 4×4096. Fig. 11b shows the fraction of misestimated packets. The misestimation fraction is within 0.01% with a sketch size of 4×4096. Note that such a sketch size only needs 32KB memory, which can be easily achieved as the memory size of modern high-speed switches is usually in the order of 10MB [45]. Besides, owing to the low estimation error, FairPolicer can fairly allocate the bandwidth among all flows. Specifically, among all simulations, the network utilization is always higher than 99.5% and the unfairness index is always lower than 0.002 (results not shown due to space limitations).

VI. Conclusion

In this paper, we find that flows using the recently proposed CCs can occupy the majority of bandwidth when contending with traditional loss-based TCP flows in a token bucket policer. We reveal this problem with various experiments. Through analysis, we find that this problem is caused by the different sensitivity of new CCs and traditional CCs to the packet loss. To tackle this problem, we propose FairPolicer — a low-latency traffic policer that can fairly allocate bandwidth among contending flows regardless of CC algorithms. Our testbed experiments show that FairPolicer can guarantee fairness among various kinds of flows.

Acknowledgment

We thank the anonymous reviewers for their constructive comments. Danfeng Shan would like to thank Yifan Liu for the proofreading. This work is supported by the National Natural Science Foundation of China (No. 61902307, 61772412, 61972421, 61702407, 61872208, U19B2025), the Innovation-Driven Project of Central South University (No. 2020CX033), the Fundamental Research Funds for the Central Universities (No. xzy012020014, xjj2018013), and the Natural Science Basic Research Plan in Shaanxi Province of China (No. 2018JM6109).

REFERENCES

[1] A. M. Kakhki, F. Li, D. Choffnes, E. Katz-Bassett, and A. Mislove, "BingeOn Under the Microscope: Understanding T-Mobiles Zero-Rating Implementation," in *ACM Internet-QoE*, 2016.

[2] T. Flach, P. Papageorge, A. Terzis, L. Pedrosa, Y. Cheng, T. Karim, E. Katz-Bassett, and R. Govindan, "An Internet-Wide Analysis of Traffic Policing," in *ACM SIGCOMM*, 2016.

[3] T. Flach, L. Pedrosa, E. Katz-Bassett, and R. Govindan, "A Longitudinal Analysis of Traffic Policing Across the Web," USC Computer Science, Tech. Rep. 15-961, 2015.

[4] F. Li, A. A. Niaki, D. Choffnes, P. Gill, and A. Mislove, "A Large-Scale Analysis of Deployed Traffic Differentiation Practices," in *ACM SIGCOMM*, 2019.

[5] H. Guo and J. Heidemann, "Detecting ICMP Rate Limiting in the Internet," in *PAM*, 2018.

[6] Cisco, "Comparing Traffic Policing and Traffic Shaping for Bandwidth Limiting," Tech. Rep., May 2014. [Online]. Available: https://www.cisco.com/c/en/us/support/docs/quality-of-service-qos/qos-policing/19645-policevsshape.html.

[7] Wu-Chang Feng, D. D. Kandlur, D. Saha, and K. G. Shin, "Understanding TCP Dynamics in an Integrated Services Internet," in *IEEE NOSSDAV*, 1997.

[8] W.-C. Feng, D. D. Kandlur, D. Saha, and K. G. Shin, "Understanding and Improving TCP Performance over Networks with Minimum Rate Guarantees," *IEEE/ACM Transactions on Networking*, vol. 7, no. 2, pp. 173–187, Apr. 1999.

[9] H. Su and M. Atiquzzaman, "Performance Modeling of Differentiated Service Network with a Token Bucket Marker," in *IEEE LANMAN*, 2001.

[10] S. Sahu, P. Nain, C. Diot, V. Firoiu, and D. Towsley, "On Achievable Service Differentiation with Token Bucket Marking for TCP," in *ACM SIGMETRICS*, 2000.

[11] R. van Haalen and R. Malhotra, "Improving TCP Performance with Bufferless Token Bucket Policing: A TCP Friendly Policer," in *IEEE LANMAN*, 2007.

[12] W. M. Zuberek and D. Strzeciwilk, "Modeling Traffic Shaping and Traffic Policing in Packet-Switched Networks," *Journal of Computer Sciences and Applications*, vol. 6, no. 2, pp. 75–81, Oct. 2018.

[13] K. Winstein and H. Balakrishnan, "TCP Ex Machina: Computer-generated Congestion Control," in *ACM SIGCOMM*, 2013.

[14] M. Dong, Q. Li, D. Zarchy, P. B. Godfrey, and M. Schapira, "PCC: Re-architecting Congestion Control for Consistent High Performance," in *USENIX NSDI*, 2015.

[15] G. Carlucci, L. De Cicco, S. Holmer, and S. Mascolo, "Analysis and Design of the Google Congestion Control for Web Real-time Communication (WebRTC)," in *ACM MMSys*, 2016.

[16] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson, "BBR: Congestion-Based Congestion Control," *ACM Queue*, vol. 14, September-October, pp. 20–53, 2016.

[17] M. Dong, T. Meng, D. Zarchy, E. Arslan, Y. Gilad, B. Godfrey, and M. Schapira, "PCC Vivace: Online-Learning Congestion Control," in *USENIX NSDI*, 2018.

[18] V. Arun and H. Balakrishnan, "Copa: Practical Delay-Based Congestion Control for the Internet," in *USENIX NSDI*, 2018.

[19] S. Abbasloo, Y. Xu, and H. J. Chao, "C2TCP: A Flexible Cellular TCP to Meet Stringent Delay Requirements," *IEEE Journal on Selected Areas in Communications*, vol. 37, no. 4, pp. 918–932, Apr. 2019.

[20] N. Jay, N. Rotman, B. Godfrey, M. Schapira, and A. Tamar, "A Deep Reinforcement Learning Perspective on Internet Congestion Control," in *ICML*, 2019.

[21] T. Meng, N. R. Schiff, P. B. Godfrey, and M. Schapira, "PCC Proteus: Scavenger Transport And Beyond," in *ACM SIGCOMM*, 2020.

[22] S. Abbasloo, C.-Y. Yen, and H. J. Chao, "Classic Meets Modern: A Pragmatic Learning-Based Congestion Control for the Internet," in *ACM SIGCOMM*, 2020.

[23] N. Garg. "Evaluating Copa Congestion Control for Improved Video Performance," Facebook. (Nov. 17, 2019), [Online]. Available: https://engineering.fb.com/video-engineering/copa/.

[24] A. Ivanov. "Evaluating BBRv2 on the Dropbox Edge Network," Dropbox. (Dec. 17, 2019), [Online]. Available: https://dropbox.tech/infrastructure/evaluating-bbrv2-on-the-dropbox-edge-network.

[25] M. Alizadeh, A. Kabbani, T. Edsall, B. Prabhakar, A. Vahdat, and M. Yasuda, "Less is More: Trading a Little Bandwidth for Ultra-low Latency in the Data Center," in *USENIX NSDI*, 2012.

[26] M. Handley, "Delay is Not an Option: Low Latency Routing in Space," in *ACM HotNets*, 2018.

[27] G. Cormode and S. Muthukrishnan, "An Improved Data Stream Summary: The Count-Min Sketch and its Applications," *Journal of Algorithms*, vol. 55, no. 1, pp. 58–75, 2005.

[28] S. Ha, I. Rhee, and L. Xu, "CUBIC: A New TCP-friendly High-speed TCP Variant," *SIGOPS Oper. Syst. Rev.*, vol. 42, no. 5, pp. 64–74, Jul. 2008.

[29] Microsoft, *Updates on Windows TCP*, https://datatracker.ietf.org/meeting/100/materials/slides-100-tcpm-updates-on-windows-tcp, Nov. 2017.

[30] *Full Proof of Theorems of PCC Vivace*, http://ttmeng.net/pubs/vivace_proof.pdf.

[31] M. Hock, R. Bless, and M. Zitterbart, "Experimental Evaluation of BBR Congestion Control," in *IEEE ICNP*, 2017.

[32] S. Claypool, "Sharing but not Caring - Performance of TCP BBR and TCP CUBIC at the Network Bottleneck," Worcester Polytechnic Institute, Tech. Rep., Mar. 2019.

[33] A. Kortebi, L. Muscariello, S. Oueslati, and J. Roberts, "Evaluating the Number of Active Flows in a Scheduler Realizing Fair Statistical Bandwidth Sharing," in *ACM SIGMETRICS*, 2005.

[34] C. Hu, Y. Tang, X. Chen, and B. Liu, "Per-Flow Queueing by Dynamic Queue Sharing," in *IEEE INFOCOM*, 2007.

[35] N. K. Sharma, M. Liu, K. Atreya, and A. Krishnamurthy, "Approximating Fair Queueing on Reconfigurable Switches," in *USENIX NSDI*, 2018.

[36] N. K. Sharma, A. Kaufmann, T. Anderson, A. Krishnamurthy, J. Nelson, and S. Peter, "Evaluating the Power of Flexible Packet Processing for Network Resource Allocation," in *USENIX NSDI*, 2017.

[37] A. K. Choudhury and E. L. Hahne, "Dynamic Queue Length Thresholds For Shared-memory Packet Switches," *IEEE/ACM Transactions on Networking*, vol. 6, no. 2, pp. 130–140, 1998.

[38] I. Rhee, L. Xu, S. Ha, A. Zimmermann, L. Eggert, and R. Scheffenegger, *CUBIC for Fast Long-Distance Networks*, RFC 8312, Feb. 2018.

[39] *Copa code*, https://github.com/venkatarun95/genericCC.

[40] *PCC Vivace code*, https://github.com/PCCproject/PCC-Uspace.

[41] R. K. Jain, D.-M. W. Chiu, and W. R. Hawe, "A Quantitative Measure of Fairness and Discrimination," Tech. Rep. DEC Research Report TR-301, Sep. 1984.

[42] A. B. and Alberto Dainotti and A. Pescapè, "A Tool for the Generation of Realistic Network Workload for Emerging Networking Scenarios," *Computer Networks*, vol. 56, no. 15, pp. 3531–3547, 2012.

[43] V. Paxson and S. Floyd, "Wide Area Traffic: the Failure of Poisson Modeling," *IEEE/ACM Transactions on Networking*, vol. 3, no. 3, pp. 226–244, Jun. 1995.

[44] *ns-3*, https://www.nsnam.org/.

[45] *Broadcom Tomahawk*, https://people.ucsc.edu/ warner/Bufs/tomahawk.