Enforcing Fairness in the Traffic Policer Among Heterogeneous Congestion Control Algorithms

Danfeng Shan, *Member, IEEE*, Linbing Jiang, Peng Zhang, Wanchun Jiang, Hao Li, Yazhe Tang, and Fengyuan Ren, *Member, IEEE*

***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: flows using new CC algorithms can easily occupy the majority of bandwidth, starving traditional TCP flows. We confirm this problem with experiments and reveal its root cause as follows. Without a buffer in traffic policers, congestion *only* causes packet losses, while new CC algorithms are *loss-resilient*. When being policed, they will not reduce the sending rate until an unacceptable loss ratio for TCP is reached, resulting in low throughput for competing TCP flows. Simply adding a 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 a token as a basic unit of bandwidth and fairly allocates tokens to active flows in a round-robin manner. To avoid bandwidth waste when flows come and go, FairPolicer puts all available tokens in a global bucket and maintains the amount of residual bucket space rather than the number of available tokens. To scale to massive concurrent flows, FairPolicer uses a Count-Min Sketch structure to maintain per-flow data with a small memory footprint. Testbed experiments show that FairPolicer can allocate bandwidth in a max-min fair manner and achieve much lower latency than other kinds of rate limiters.**

***Index Terms*— Internet, traffic policing, congestion control, packet loss.**

Manuscript received 24 November 2021; revised 7 May 2022 and 9 November 2022; accepted 16 April 2023; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor J. Llorca. Date of publication 16 June 2023; date of current version 16 February 2024. This work was supported in part by the National Natural Science Foundation of China under Grant 61902307, Grant 62272382, Grant 61972421, Grant 62172323, and Grant 62002165, and in part by the Natural Science Foundation of Jiangsu Province under Grant BK20200445. The preliminary version of this paper was published in the Proceedings of IEEE INFOCOM 2021 [DOI: 10.1109/INFOCOM42981.2021.9488761]. *(Corresponding author: Peng Zhang.)*

Danfeng Shan, Linbing Jiang, and Yazhe Tang are with the School of Computer Science and Technology, Xi'an Jiaotong University, Xi'an 710049, China (e-mail: dfshan@xjtu.edu.cn; jlb337@stu.xjtu.edu.cn; yztang@xjtu.edu.cn).

Peng Zhang and Hao Li are with the MOE Key Laboratory for Intelligent Networks and Network Security, School of Computer Science and Technology, Xi'an Jiaotong University, Xi'an 710049, China (e-mail: p-zhang@xjtu.edu.cn; hao.li@xjtu.edu.cn).

Wanchun Jiang is with the School of Information Science and Engineering, Central South University, Changsha 410083, China (e-mail: jiangwc@csu.edu.cn).

Fengyuan Ren is with the Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China (e-mail: renfy@tsinghua.edu.cn).

This article has supplementary downloadable material available at https://doi.org/10.1109/TNET.2023.3276410, provided by the authors.

Digital Object Identifier 10.1109/TNET.2023.3276410

I. INTRODUCTION

ON THE Internet, an Internet Service Provider (ISP) usually needs to regulate its customers' traffic rate to enforce various network policies [1], [2], [3], [4], [5], [43]. 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 a 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], [2], [3].

The impact of traffic policing on network performance has been studied extensively in literature [7], [8], [9], [10], [11], [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], [14], [15], [16], [17], [18], [19], [20], [21], [22], [23], [24]. 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 [16], [17], [25], [26], [27] 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 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 a congestion signal, 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 on today's Internet demand ultra-low latency [28], [29]. 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 service providers tend to deliver their data as fast as possible and may not be willing to concede bandwidth for the purpose of fairness.

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 the 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, making 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 waste of bandwidth. Second, with many concurrent flows, maintaining per-flow data requires lots of memory.

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 [30] to maintain per-flow residual bucket space, 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 10.3× and 26.9× compared to token bucket policer and traffic shaper, respectively.

In sum, our contribution is three-fold:

- We 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 fairness while still achieving a much lower latency compared with other kinds of rate limiters.

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

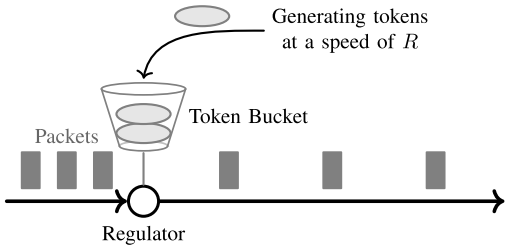


Fig. 1. Token Bucket Policer.

The rest of the paper proceeds as follows: Section II briefly introduces traffic policing and congestion control algorithms. Section III demonstrates and analyzes the unfairness problem. Section IV describes FairPolicer in detail. The implementation and evaluation are presented in Section V. The related work is summarized in Section VI. Finally, the paper is concluded in Section VII.

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

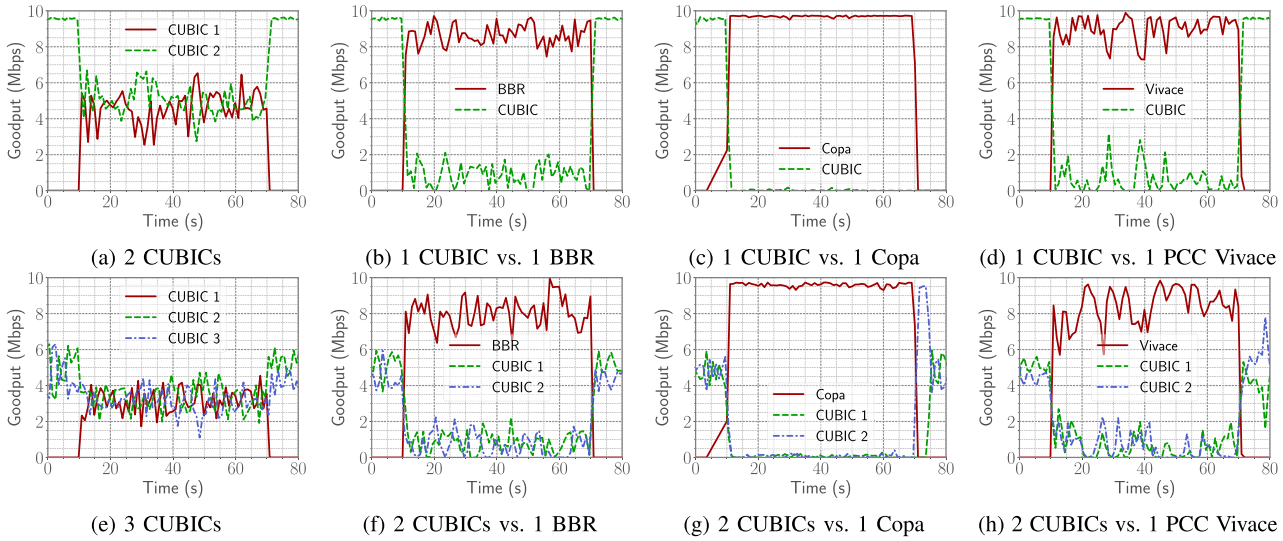
Traffic policing is widely used by ISPs to enforce a specific traffic rate. For example, T-Mobile's "Binge On" service provides zero-rated access to video streaming while limiting the traffic rate to 1.5Mbps with a traffic policer [1]. Some other ISPs use traffic policing to enforce their customers' 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 simplicity, one can easily deploy many 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 [29].

B. Congestion Control Algorithms

Congestion Control (CC) is crucial to the user experience of Internet applications. Currently, CUBIC [31] — the default CC in Linux and Windows [32] — is the most popular CC algorithm on the Internet [27]. 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 leads to flows experiencing very high latency. On the other hand, CUBIC cannot distinguish



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

(e) 3 CUBICs (f) 2 CUBICs vs. 1 BBR (g) 2 CUBICs vs. 1 Copa (h) 2 CUBICs 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.

between congestion-induced packet loss and non-congestion-induced packet loss (e.g., random packet loss), leading to degradation of throughput in some lossy scenarios.

Due to the above reasons, lots of new CC algorithms have been proposed recently [13], [14], [16], [17], [19], [20], [21], [22], [23], [24]. In addition to packet loss, these CC algorithms also incorporate other network feedback to achieve low latency and high throughput simultaneously. For example, BBR [17] maintains an estimate of the round-trip propagation time (i.e., round-trip time (RTT) 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 has been deployed on YouTube video servers since 2016 [17]. Copa [20] 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 on Facebook for live video upload [25]. PCC Allegro [14] and PCC Vivace [19] regard the network as a black box and adjust the sending rate to maximize a utility value, which 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-B). One host serves as the receiver and other hosts serve as senders. The traffic rate is throttled to 10Mbps by a 50KB token bucket. The base RTT between hosts is 10ms.

Fig. 2 shows the evolution of goodput when different kinds of flows are rate-limited by the same token bucket policer.

TABLE I
Unfairness Problem Between CUBIC and New CCs

Congestion Control Algorithms	Fraction of Occupied Bandwidth
BBRv2	77.5%
PCC Allegro	89.1%
Verus	96.7%
Indigo	94.1%
RemyCC	98.3%
FillP	99.1%
FillP-Sheep	98.3%

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 also be observed with Copa and PCC (Fig. 2c and Fig. 2d). Specifically, competing with a CUBIC flow, the BBR, Copa, and PCC Vivace flow can occupy 90.6%, 99.8%, and 93.8% of bandwidth, respectively. Competing with two CUBIC flows, the BBR, Copa, and PCC Vivace flow can occupy 84.2% (Fig. 2f), 98.8% (Fig. 2g), and 88.2% of bandwidth (Fig. 2h), respectively.

We have also tested some other new CC algorithms and found a similar phenomenon. Table I summarizes the fraction of throughputs occupied by the new CC with one CUBIC flow and one new CC flow rate-limited by the same policer.

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 bandwidth in 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

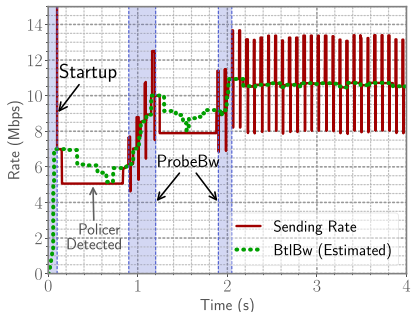


Fig. 3. Temporal behavior of BBR.

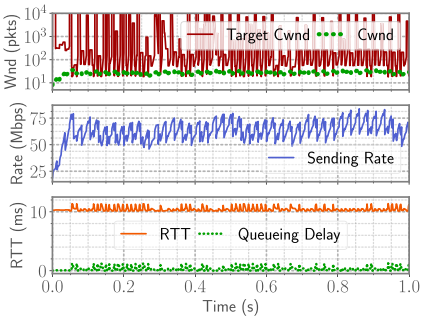


Fig. 4. Temporal behavior of Copa.

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 (the total bandwidth is 10Mbps).

Second, even if the BBR flow has not occupied the majority of 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 by 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 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

[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.

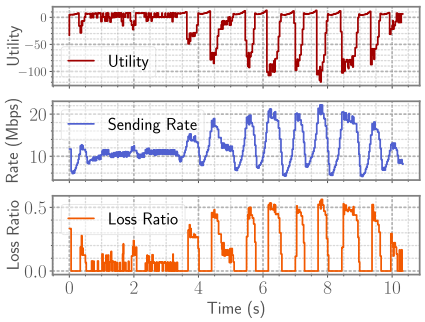


Fig. 5. Temporal behavior of PCC Vivace.

Copa is smaller than 0.1ms most of the time. Consequently, 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 bandwidth, starving the CUBIC flow.

3) PCC Vivace: PCC Vivace adjusts its sending rate according to a utility function. The 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, 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 [19] and [33], 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% [19]. As a result, the majority of 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 tolerance 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.

IV. Mitigating the Unfairness Problem

In this section, we first discuss the limitations of existing ways to mitigate the unfairness problem. Then we describe the design of FairPolicer.

A. Possible Solutions

1) Replacing Policing With Traffic Shaping: Traffic shaping is another approach for 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 congested. 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 shaper, resulting in high in-network latency. This is quite unacceptable as latency is the limiting factor for many Internet applications today [29], 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 [34], [35]. 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.

2) Improving CC at Senders: Another way is to improve CC at senders to fairly compete with other loss-based CCs. For example, we can add a policer detector at the sender.[3]If the sender finds itself being rate-limited by a token bucket policer, it can switch its CC to a loss-based mode to fairly compete with other loss-based CCs.

However, we think that it is difficult to put this solution into action. Specifically, a traffic sender can either be a service provider or an Internet user. On the one hand, it is not easy to persuade all service providers to fairly compete for bandwidth with other providers, as a higher throughput denotes a better user experience, which can improve the revenue of enterprises. On the other hand, most of CCs are implemented in the operating system. Upgrading the operating system is not easy for individuals [36].

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 one cause of the unfairness problem is a lack of bandwidth allocation mechanisms in the current traffic policing scheme. Specifically, tokens are blindly 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 into 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.[4]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.

[3]The policer detector in BBR is insufficient as it can only detect the situation that a single BBR flow is policed.

[4]The way to distinguish a flow depends on the intent of the network operator. For example, if the network operator wants to enforce fine-grained fairness among all connections, a flow can be identified by a 5-tuple (i.e., source IP address, destination IP address, source TCP/UDP port, destination TCP/UDP port, and IP protocol). On the other hand, if the network operator only wants to enforce fairness among different network users, a flow can be identified by the destination IP address.

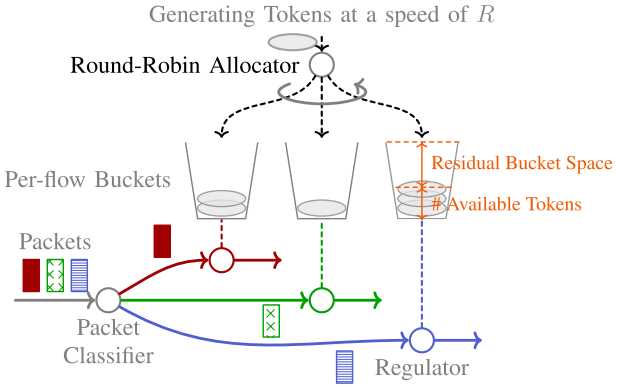


Fig. 6. Basic Idea.

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 being 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 [37], [38]. Accurately maintaining the 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 (as shown in Fig. 6). The residual bucket space is the free bucket space for tokens (i.e., residual bucket space = bucket capacity − # tokens). It denotes the number of *occupied tokens* of each flow. For example, if the total bucket capacity of a flow is 100 tokens and the bucket contains 10 available tokens now, the residual bucket space is 90 tokens, which denotes that 90 tokens are occupied by this flow.

Allocating a token to a flow can be achieved by decreasing its residual bucket space. In this way, FairPolicer can be adaptive to the variation of active flows. Specifically, when a flow becomes inactive, its bucket will be full of tokens and its residual bucket space will become empty. FairPolicer can simply delete the bucket and reallocate its tokens by raising the bucket capacity of other flows. When a flow becomes active, its residual bucket space is empty 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.

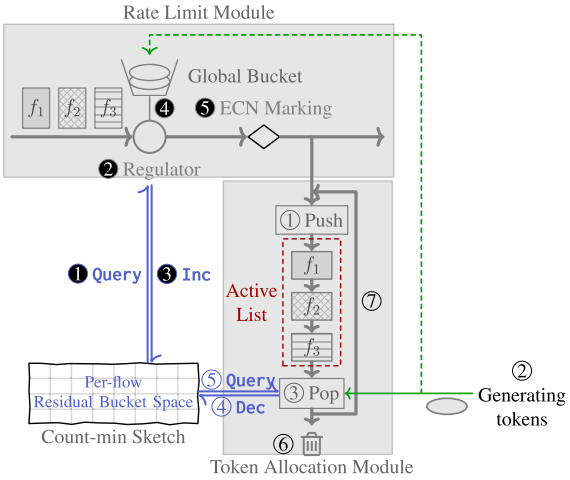


Fig. 7. Structure of FairPolicer.

Second, we use the Count-Min Sketch structure [30] to estimate the residual bucket space 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 are pairwise independent. A Count-Min Sketch supports two operations. (1) `Update(f,c)` updates the counter of flow f by c, where 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 [39]. Some recent studies have shown that Count-Min Sketch can track per-flow statistics in sublinear space [39], [40]. In §IV-E and §V-H, we'll further show that Count-Min Sketch can achieve high estimation accuracy with a small memory footprint.

D. Details of FairPolicer

Fig. 7 depicts 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. First, 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.

Second, FairPolicer limits the per-flow traffic rate by restricting the per-flow residual bucket space with a threshold. The threshold represents the token bucket capacity of each flow. To ensure fairness, all flows use the same threshold. Formally, an arriving packet of flow i is allowed to pass the regulator if and only if

$$\theta^{(i)} < T_{token} \tag{3}$$

where $\theta^{(i)}$ is the residual bucket space for flow i and T is the token threshold. As mentioned before, the threshold is adjusted according to the number of active flows. With n active flows, FairPolicer sets the threshold as

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

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 $B/(n+1)$ tokens so that newly arriving flows will not starve for tokens. Second, except for the reserved ones, the remaining tokens can be equally distributed among the active flows.

One may argue that reserving tokens 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 [41] to approximately adjust the threshold as in (4). Specifically, FairPolicer sets the threshold as

$$T_{token} = k = B - \sum_{i=1}^{n} \theta^{(i)} \tag{5}$$

where k is the number of available tokens in the global bucket. 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_{token} 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, $\theta^{(1)} = \theta^{(2)} = \cdots = \theta^{(n)} = T_{token}$. According to (5), the threshold can be given by

$$T_{token} = B - \sum_{i=1}^{n} \theta^{(i)} = B - nT_{token} \tag{6}$$

which is equivalent to (4).

In this way, FairPolicer only needs to maintain the number of available tokens, which is very easy to be implemented.

Third, FairPolicer utilizes ECN marking to enrich the congestion feedback. Current traffic policer only generates packet loss as a congestion signal, this may cause a high packet loss ratio [2]. Borrowing the idea from active queue management, FairPolicer generates ECN signals to explicitly notify senders of congestion when the tokens are about to be exhausted. Specifically, FairPolicer marks packets with ECN if and only if

$$\theta^{(i)} \geqslant T_{ecn} \tag{7}$$

where T_{ecn} is the marking threshold and $T_{ecn} < T_{token}$. When receiving the ECN signal, senders will reduce their sending rates without exhausting the tokens in FairPolicer. In this way, flows can adjust their sending rates without suffering packet drops.

Combining the above mechanisms, the rate limit module works as follows. When a packet arrives, FairPolicer reads the residual bucket space of the corresponding flow (denoted by $\theta^{(i)}$) from the Count-Min Sketch (❶). The packet is

allowed to pass through the regulator if and only if $\theta^{(i)} < k$ (❷), where k is the number of tokens in the global bucket. If the packet is allowed to pass through the regulator, there are three subsequent operations. First, FairPolicer increases the residual bucket space by updating the Count-Min Sketch (❸). Second, FairPolicer reduces the number of available tokens in the global bucket by decreasing k (❹). Third, if $\theta^{(i)} \geqslant T_{ecn}$, FairPolicer marks packets with ECN (❺).

The token allocation module fairly allocates the generated tokens to the active flows. The core of this module is maintaining an active flow list to allocate tokens in a round-robin manner. 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.

We observe that there is no need to maintain a "real" active flow list. FairPolicer only needs to maintain a list of flows whose residual bucket space is larger than zero, because FairPolicer does not need to allocate tokens to the flows whose buckets have already been filled with tokens. FairPolicer considers a flow as active if and only if its residual bucket space is larger than zero. Specifically, a flow becomes active if its residual bucket space becomes above zero, while a flow becomes inactive if its residual bucket space becomes zero.

To maintain such an active flow list, FairPolicer adds a flow into the list whenever its residual bucket space becomes above zero, and removes a flow from the list whenever its residual bucket space 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 residual bucket space before the packet arrival is zero (①). The latter operation is conducted after a token is generated. Whenever generating a token (②), FairPolicer pops a flow from the list (③) and allocates the token to the flow by decreasing its residual bucket space (④). Then FairPolicer queries the Count-Min Sketch to check whether the residual bucket space of this flow is zero (⑤). If so, FairPolicer removes this flow from the list (⑥). Otherwise, FairPolicer appends the flow to the end of the list (⑦).

In addition, this module is also responsible for generating tokens. If the throttling rate is R and the size of a token is 1 byte, then tokens are generated every $1/R$ seconds. However, with a high throttling rate (e.g., 10Gbps), the tokens should be generated at high speed, triggering the subsequent operations at a very high frequency. This can incur high overhead for the policing device. To reduce the overhead in such environments, FairPolicer can generate tokens at a coarser granularity. Specifically, FairPolicer can generate several tokens, say, 1KB at a time, and allocate all of them to a flow. In this way, the frequency of operations can be reduced by 1000×. Such a method trades off bandwidth. Specifically, if a flow occupies less than 1KB tokens, some generated tokens are wasted. However, we argue that the bandwidth waste is negligible as long as the number of generated tokens is far smaller than the flow's bucket capacity.

E. Accuracy of Count-Min Sketch

Count-Min Sketch can over-estimate a flow's residual bucket space 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 [30].

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 three kinds of parameters: bucket capacity B, ECN threshold T_{ecn}, 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 analyze the setting of bucket capacity B and ECN threshold T_{ecn}. Besides, we focus on analyzing the parameter settings for the CUBIC flows. This is because loss-resilient flows can always achieve high throughput when passing through FairPolicer and are not sensitive to these parameters. Furthermore, CUBIC is the most common loss-sensitive CC algorithm on the Internet [27].

Theorem 2: To guarantee that ECN-enabled CUBIC flows can fully utilize the allocated bandwidth and avoid packet drops, the token threshold T_{token} and ECN threshold T_{ecn} should meet the following requirements:

$$T_{token} \geqslant \begin{cases} 4T_{ecn}, & T_{ecn} < W_0 2^{M-1} \\ 6T_{ecn} - \frac{RD}{n}, & W_0 2^{M-1} \leqslant T_{ecn} < W_0 2^{M} \\ 7T_{ecn} - \frac{2RD}{n}, & T_{ecn} \geqslant W_0 2^{M} \end{cases} \tag{8}$$

$$T_{ecn} \geqslant \frac{G}{D} \cdot (\frac{RD}{n})^{\frac{4}{3}} \tag{9}$$

where W_0 denotes the initial congestion window, R denotes the throttling rate, D denotes the round-trip time, n denotes the number of active flows, $G = \frac{3}{4\sqrt[3]{C}} \left(\frac{\beta}{4-\beta}\right)^{\frac{4}{3}}$, β is the window reduction factor of CUBIC, and $M = \max\left\{\left\lceil \log_2 \frac{RD}{W_0} \right\rceil, 0\right\}$.

Proof: See Appendix A. ■

Remarks: The above parameter settings are non-trivial in practical, as it relies on the number of active flows, which is a changing variable. Thus, we suggest setting the parameter as follows.

First, FairPolicer can dynamically adjust T_{ecn} according to T_{token}. Specifically, we can set

$$T_{ecn} = \frac{T_{token}}{8} \tag{10}$$

This can ensure the satisfaction of (8). Furthermore, T_{ecn} can be easily computed by shifting T_{token}.

Second, as $T_{token} = \frac{B}{n+1} = 8T_{ecn}$, inequality (9) can be rewritten as

$$B \geqslant \left(\frac{1}{n^{\frac{1}{3}}} + \frac{1}{n^{\frac{4}{3}}}\right) \frac{8G}{D} (RD)^{\frac{4}{3}} \tag{11}$$

Note that the right part reaches its maximum at $n = 1$. Thus, we have $B \geqslant \frac{16G}{D}(RD)^{\frac{4}{3}}$. On the other hand, the

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

$$B = \frac{16G}{D}(RD)^{\frac{4}{3}} \tag{12}$$

According to the RFC [42], the parameters of CUBIC should be set as $\beta = 0.3$ and $C = 0.4$. Thus, we have $G = 0.036$. Substituting it into (12) yields $B = \frac{0.576}{D}(RD)^{\frac{4}{3}}$.

Finally, to determine B in Eq. (12), we need to acquire D (i.e., RTT). There are two considerations. First, measuring RTT on the fly can incur high overhead. Thus, we should measure RTT and set the parameter offline. Second, various flows tend to have different RTTs. To ensure that all flows can fully utilize the bandwidth, we should set D as the maximum RTT of all flows in the network. For the network whose RTT distribution is extremely long-tailed, we can choose a high-percentile RTT (e.g., 90th percentile) such that the majority of flows can fully utilize the bandwidth while the bucket size is not too large.

V. Implementation and Evaluation

In this section, we evaluate FairPolicer with testbed experiments and ns-3 simulations. The result highlights include:

- FairPolicer can fairly allocate bandwidth to contending flows with various CCs, including CUBIC, BBR, Copa, PCC Vivace, and aggressive UDP (§V-C, §V-D, §V-E, §V-F, and §V-K).
- FairPolicer can reduce the average web page load latency by up to 6.6× and the tail web page load latency by up to 26.9× (§V-G and §V-K).
- FairPolicer can achieve a misestimation fraction of 0.001 using only 32KB memory with 1K-2K concurrent flows (§V-H).

A. Implementation

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 manually set the maximum queue size to one packet to emulate a bufferless policer. Different from the `tc-tbf qdisc`, which enforces traffic rate in the `dequeue` function, we implement all FairPolicer functions in the `enqueue` function. This is because aggressive flows may dominate the queue and starve conservative flows when all functions are realized in the `dequeue` function, resulting in the inability of FairPolicer to enforce fair bandwidth sharing.

Algorithm 1 shows the pseudocode of the `enqueue` function. Tokens are generated and allocated to the active flows at the beginning (line 3-5). Then FairPolicer decides whether to enqueue this packet according to the residual bucket space of the corresponding flow (line 6-7). If a packet is allowed to be enqueued, FairPolicer reduces the number of available tokens and updates the residual bucket space for the flow (line 8-9). FairPolicer marks the packet with ECN if the residual bucket space is larger than T_{ecn} (line 10-11). If this flow is inactive, FairPolicer pushes this flow into the active queue (line 12-13). Finally, the packet is enqueued to the internal queue of the qdisc (line 14). The packets in the internal queue are sent to the NIC through the qdisc's `dequeue` function.

Algorithm 1 Packet Enqueue

Inputs: Bucket capacity B, enforcing rate R, Count-Min Sketch cms, active flow list $active\_list$, packet to be enqueued pkt, current time t, number of available tokens k, previous time of running this function $prev\_t$, internal queue length $queue\_length$.

```
1  if queue_length > 1 then
2      Drop pkt;
   // Generate tokens
3  toks ← R · (t − prev_t);
4  k ← min(k + toks, B);
   // Allocate tokens to active flows
5  allocate_tokens(token_num);
   // Estimate residual bucket space
6  θ ← cms.query(pkt.flow_id);
7  if θ < k and pkt.length ⩽ k then
8      k ← k − pkt.length;
       // Increase the residual bucket
          space
9      cms.update(pkt.flow_id, pkt.length);
10     if θ >= T_ecn then
11         Mark packet with ECN;
12     if θ = 0 then // A new flow becomes
        active
13         active_list.push(pkt.flow_id);
14     Enqueue pkt into the internal queue of the qdisc;
15 else
16     Drop pkt;
17 prev_t ← t;

18 Function allocate_tokens(toks):
19     while toks > 0 and not active_list.empty() do
20         flow_id ← active_list.pop();
21         cmd.update(flow_id, -1);
22         θ ← cms.query(flow_id);
23         if θ > 0 then // Flow is still active
24             active_list.push(flow_id);
25         toks ← toks − 1;
```

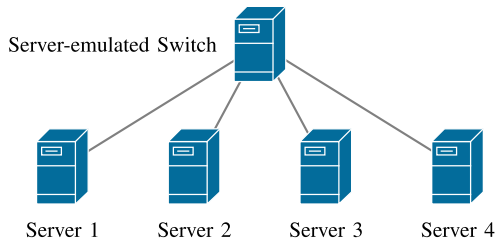
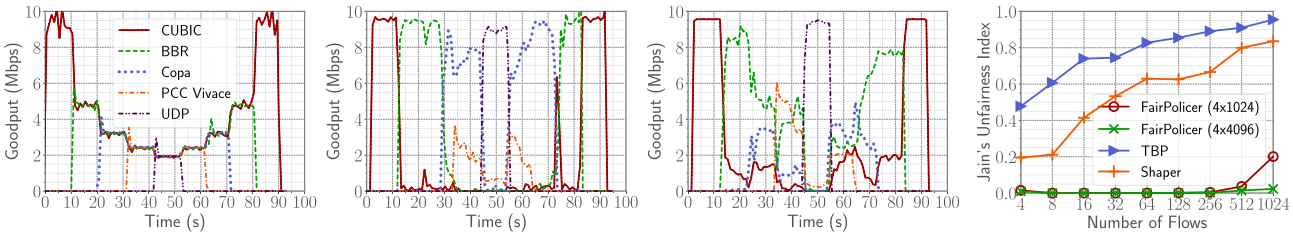


Fig. 8. Topology of our testbed.

B. Testbed Setup

We build a testbed with four servers connected by another server emulating a switch (as shown in Fig. 8). 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,



(a) Temporal fairness: FairPolicer (b) Temporal fairness: TBP (c) Temporal fairness: shaper (d) Unfairness index

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

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 20.04 with Linux kernel 5.4.0. We use `tc-netem` to enlarge the RTT to 20ms.

In the server-emulated switch, we consider three kinds of rate limiters: FairPolicer, token bucket policer (TBP), and traffic shaper. These rate limiters are deployed 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 1790KB according to Eq. (12).[5] The ECN threshold is set as in Eq. (10). The token granularity is 100B. 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 111KB according to the analysis in our previous work [43]. For traffic shaper, we set the bucket size to 50KB and the queue size to 111KB. The throttling rate of each rate limiter 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 5.4.0. We use `iperf3` to generate CUBIC and BBR traffic. We use the user-space implementation of Copa [44] and PCC Vivace [45] 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.

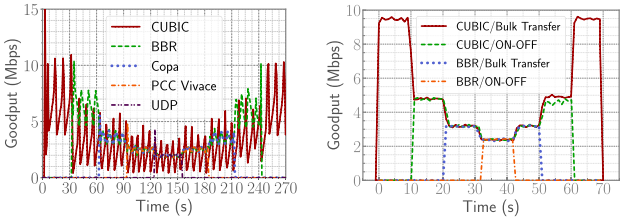
C. Fairness Among Various CCs

Temporal Fairness: First, we evaluate the fairness of bandwidth allocation among a CUBIC flow, a BBR flow, a Copa flow, a PCC Vivace flow, and an aggressive UDP flow. The UDP flow is ill-behaved that blindly sends traffic at line rate (i.e., nearly 1Gbps). 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. In comparison, these heterogeneous CCs cannot fairly share bandwidth with TBP (Fig. 9b) and traffic shaper (Fig. 9c).

Fairness Index: Next, we use Jain's fairness index [46] to quantitatively evaluate the goodput fairness of FairPolicer, TBP, and traffic shaper. Jain's fairness index is given by

$$\frac{\left(\sum_i x_i\right)^2}{n \cdot \sum_i x_i^2} \tag{13}$$

[5]Note that in Eq. (12) the unit of D is seconds and the unit of R is MSS (Maximum Segment Size) per second.



(a) Different RTTs (b) Different traffic types

Fig. 10. Fairness with different RTTs and traffic types.

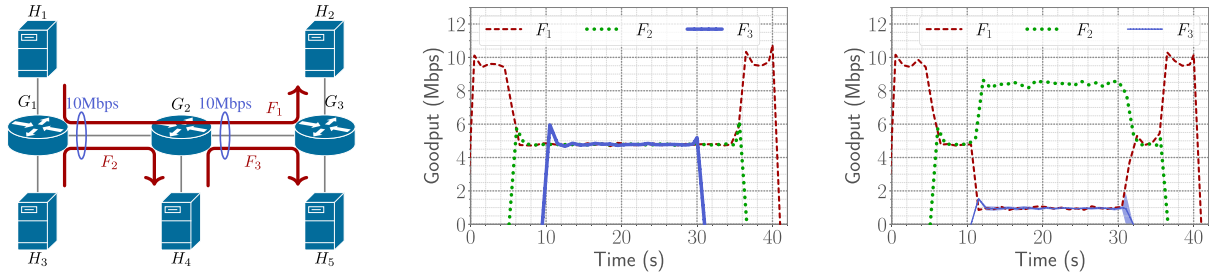
where x_i denotes the throughput of the i-th flow. The fairness index ranges from $1/n$ (worst case) to 1 (best case), where n is the number of flows.

In this experiment, we start the same number of CUBIC, BBR, Copa, and PCC Vivace flows simultaneously. Fig. 9d shows 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 0.02 when the number of concurrent flows is within the sketch size. In comparison, the unfairness indexes of TBP and traffic shaper are 0.61-0.95 and 0.21-0.84, 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 analyzed in §IV-E, 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 1K flows.

D. Fairness With Different RTTs

In this part, we evaluate the fairness among flows with different RTTs. The RTT of the CUBIC flow is 100ms, while the RTTs of other flows are 10ms. Other settings are kept unchanged. Fig. 10a shows the goodput evolution of each flow. FairPolicer can fairly allocate bandwidth among BBR, Copa, PCC Vivace, and UDP flows. For the CUBIC flow, its goodput fluctuates significantly. This is because the CUBIC flow has a large congestion window with a long RTT. Consequently, the amplitude and the period of its congestion window evolution are larger and longer, respectively. As a result, when competing with other flows, the goodput of the CUBIC flow is slightly smaller. Nevertheless, FairPolicer can effectively avoid bandwidth starvation for the CUBIC flow. Achieving perfect fair bandwidth sharing in this scenario needs to improve the congestion control algorithm at end hosts, which is beyond the scope of FairPolicer.



(a) Topology (b) Scenario 1 (c) Scenario 2

Fig. 11. Goodput evolution with multiple bottlenecks.

E. Fairness Under Different Traffic Types

In the previous experiments, we only consider one type of traffic. To complement the evaluation, we further evaluate the fairness of FairPolicer among different types of traffic. Specifically, we consider two types of traffic. One is bulk transfer traffic that is used in previous experiments. This kind of traffic is common on applications that transmit a large volume of data (e.g., file download). The other is ON-OFF traffic, which is also very common on many popular Internet applications such as video streaming. Bulk transfer traffic is generated in the same way as in previous experiments. To generate ON-OFF traffic, we let a sender transmit 256KB data periodically, which emulates YouTube streaming traffic [47]. The interval time between two successive transmissions follows an exponential distribution. The average traffic rate is 10Mbps. To evaluate fairness among different types of traffic using different types of CCs, we generate four flows: (i) a bulk-transfer flow using CUBIC, (ii) an ON-OFF flow using CUBIC, (iii) a bulk-transfer flow using BBR, and (iv) an ON-OFF flow using BBR. Other settings are kept unchanged.

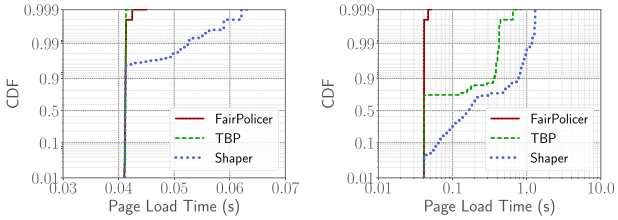
Fig. 10b shows the goodput evolution of each flow. FairPolicer can fairly allocate bandwidth among flows with different traffic patterns using different CCs. This is because FairPolicer can evenly allocate tokens to different flows. In particular, though the ON-OFF flows do not always generate traffic as bulk-transfer flows, they can achieve the same goodput. This is because the tokens generated for them are accumulated in their dedicated buckets during their OFF periods and can be used for further transmissions.

F. Fairness With Multiple Bottlenecks

We create the topology in Fig. 11a to evaluate the performance of FairPolicer with multiple bottlenecks. Five hosts (H_1 - H_5) are connected through three gateways (G_1 - G_3). There are three flows. ① F_1 is from host H_1 to host H_2; ② F_2 is from host H_3 to host H_4; ③ F_3 is from host H_4 to host H_5. There are two bottlenecks. One is between G_1 and G_2, the other is between G_2 and G_3. Flow F_2 and Flow F_3 experience one bottleneck, while Flow F_1 experiences both bottlenecks. We deploy FairPolicer at two locations: ① On the NIC of G_1 connected to G_2; ② On the NIC of G_2 connected to G_3. The throttling rate is set to 10Mbps. The base RTT of F_1 is 20ms, while the base RTT of F_2 and F_3 is 10ms. All flows use CUBIC as their CC algorithms. We evaluate FairPolicer in two scenarios.

Scenario 1: In this scenario, we start F_1, F_2, and F_3 one after another at an interval of 5s. Fig. 11b shows the evolution of goodput. When all flows are started, the goodputs of F_1, F_2, and F_3 are all around 5Mbps. Thus, FairPolicer can fairly allocate bandwidth to three flows, despite the disparities of RTTs and the number of experienced bottlenecks.

Scenario 2: In this scenario, we increase the number of F_3 flows to 9 so that the bandwidth between G_2 and G_3 is shared among 10 flows. Other settings are kept unchanged. Fig. 11c shows the evolution of goodput. We make two observations. First, FairPolicer can fairly allocate the bandwidth between G_2 and G_3 among 10 flows. Specifically, F_1 and F_3 flows achieve the same bandwidth of 1Mbps. Second, FairPolicer can efficiently allocate the bandwidth between G_2 and G_3. Specifically, while F_1 is only able to send data at 1Mbps, FairPolicer can allocate all residual bandwidth (i.e., 9Mbps) to F_2. In summary, this scenario demonstrates that FairPolicer can allocate bandwidth in a max-min manner.



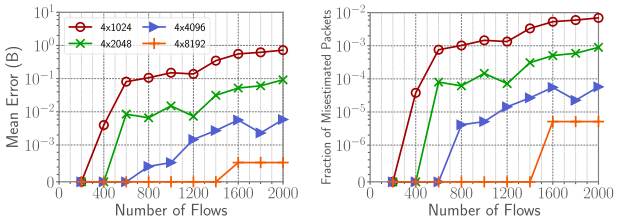
(a) Light-load background traffic (Pareto scale = 0.2ms) (b) Heavy-load background traffic (Pareto scale = 0.1ms)

Fig. 12. Distributions of web page load time.

G. 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 algorithms. We set up an NGINX web server at a sender and let a receiver periodically request a 10KB web page. The requests follow a Poisson process with an average rate of 1 request per second. We use D-ITG [48] to generate the background traffic. The inter-arrival time of packets follows a Pareto distribution with a shape parameter of 0.9 [49]. We vary the scale parameter to change the traffic load. The experiment lasts for 1,000 web requests.

Fig. 12 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. 12a), over 90% of the web requests can be finished within 42ms (i.e., ∼2 RTTs) for all kinds of rate limiters. However, traffic shaper has a longer



(a) Mean error (b) Fraction of misestimation

Fig. 13. Accuracy of Count-Min Sketch.

tail page load time than TBP and FairPolicer. Specifically, the 99th-percentile page load time of traffic shaper is 52.5ms, which is over 27.1% longer than that of TBP and FairPolicer (41.3ms). This is because the background traffic can cause queue buildup in the traffic shaper and web requests may experience some queueing latency. In comparison, TBP and FairPolicer do not contain a queue and thus can achieve low latency.

With heavy-load background traffic (Fig. 12b), FairPolicer achieves much shorter page load time than traffic shaper and TBP. Specifically, FairPolicer can achieve 2.5× and 6.6× shorter average page load time than TBP and traffic shaper, respectively. And FairPolicer can achieve 10.3× and 26.9× shorter 99th-percentile page load time than TBP and traffic shaper, respectively. This is because TBP and traffic shaper can drop some packets of web requests due to the lack of tokens and buffer space, respectively, resulting in packet retransmissions and retransmission timeouts. In comparison, FairPolicer can isolate the web request traffic and background traffic, effectively avoiding packet drops of web request flows.

H. Accuracy of Count-Min Sketch

In this part, we use ns-3 simulations [50] to evaluate the accuracy of Count-Min Sketch with numerous 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. 13a shows the mean estimation error of per-flow residual bucket space 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. 13b 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 [51].

I. Effect of Token Granularity

As discussed in §IV-D, generating a number of tokens at a time can reduce the overhead of FairPolicer at the price of wasting some tokens and bandwidth. In this part, we evaluate the effect of token granularity on CPU overhead and bandwidth utilization. We raise the throttling rate to 200Mbps to raise the overhead of FairPolicer. The bucket size is set to 75MB according to Eq. (12). The granularity of a token is set to 1B, 1KB, 10KB, 100KB, and 1MB, respectively. We simultaneously generate 8 flows, including 2 CUBIC flows, 2 BBR flows, 2 Copa flows, and 2 PCC Vivace flows. Other settings are kept unchanged.

TABLE II
EFFECT OF TOKEN GRANULARITY

Granularity	CPU Load	Goodput (Mbps)	Unfairness Index
1B	2.0	152.89	0.54
10B	0.13	191.48	3.32e-5
100B	0.12	191.34	2.87e-6
1KB	0.12	191.33	2.94e-6
10KB	0.085	191.36	1.96e-6
100KB	0.067	191.29	3.23e-6
1MB	0.063	190.86	2.59e-5

TABLE III
EFFECTIVENESS OF ECN MARKING

Rate Limiter	Retrans Rate (1 Flow)	Retrans Rate (10 Flows)
FairPolicer w/ ECN	0.00%	0.55%
FairPolicer w/o ECN	1.97%	18.82%
TBP	1.89%	16.52%
Shaper	0.05%	2.02%

Table II shows the CPU load, overall goodput, and the unfairness index with different token granularities. With 1B granularity, the CPU load is so high that FairPolicer is unable to fully utilize and fairly allocate the bandwidth. With a granularity coarser than 10B, the CPU load is significantly reduced. As a result, FairPolicer can fully utilize and fairly allocate the bandwidth. Even when the granularity is as coarse as 1MB, the wasted bandwidth is within 0.6%, which is negligible. Thus, it is reasonable to reduce the overhead of FairPolicer by using a coarse token granularity.

J. Effectiveness of ECN Marking

To prevent a high packet loss ratio, FairPolicer marks packets with ECN when tokens are about to be exhausted. To evaluate the effectiveness of ECN marking, we respectively start 1 and 10 CUBIC flows from the same sender to the same receiver. Table III shows the retransmission rate. Indeed, without ECN marking, FairPolicer and TBP can incur a high packet loss rate. However, ECN marking can effectively reduce the packet loss rate. Specifically, with 1 flow, ECN marking can avoid all packet drops for FairPolicer. With 10 flows, ECN marking can reduce the retransmission rate by 34×. Compared to traffic shaping, FairPolicer can achieve over 3.6× lower packet retransmission rate.

K. FairPolicer on the Internet

To complement our previous evaluations on a closed and controlled testbed, we evaluate the performance of FairPolicer on the Internet. We build a testbed with 5 senders and 1 receiver. The receiver is located at Xi'an (China). The senders are located at Beijing (China), Shanghai (China), Hong Kong (China), Seoul (Korean), and Oregon (US). The RTTs between the senders and the receiver are ∼35ms (Beijing-Xi'an), ∼39ms (Shanghai-Xi'an), ∼42ms (Hong Kong-Xi'an), ∼120ms (Seoul-Xi'an), and ∼200ms (Oregon-Xi'an), respectively. The receiver is connected to the Internet via a server-emulated router. Rate limiters are deployed on the router, throttling the traffic rate to 10Mbps.

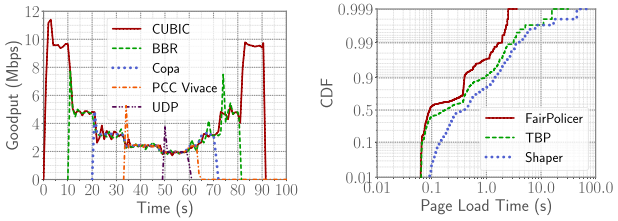


Fig. 14. Internet performance. (a) Fairness and convergence. (b) Web page load time. The background traffic follows a Pareto distribution with a scale of 0.1 ms.

Experiment 1 (Fairness): The experiment settings are the same as those in §V-C except that the CUBIC flow, BBR flow, Copa flow, PCC Vivace flow, and UDP flow originate from Beijing, Shanghai, Hong Kong, Seoul, and Oregon, respectively. Fig. 14a shows the goodput of each flow. FairPolicer can guarantee a fair allocation of bandwidth among different kinds of CCs on the Internet.

Experiment 2 (Web Page Load Latency): The experiment settings are the same as those in §V-G except that the web servers are located at 5 different locations. Fig. 14b shows the CDF of web page load time. Among three kinds of rate limiters, FairPolicer can achieve the shortest page load time. Specifically, the average page load time is 1.8× and 3.2× shorter than those of TBP and traffic shaper, respectively.

VI. Related Work

Measuring and modeling traffic policing. Some work [1], [2], [3] studies traffic policing through measurements. Kakhki et al. find that T-Mobile's "*Binge On*" service utilizes a token bucket policer to throttle the video flows to 1.5Mbps [1]. Flach et al. present an analysis of traffic policing at Google's CDN and find that up to 7% of connections are policed [2]. They also analyze the M-Lab NDT dataset and find similar results [3]. Their work focuses on the case with a single flow policed, while we study the fairness with several flows policed together. Some other work [10], [12] proposes theoretical models to analyze the performance of traffic policers. Sahu et al. propose an analytical model to determine the sending rate of a TCP flow with token bucket policing [10]. Zuberek et al. model the traffic policer with Petri net and analyze the impact of traffic policing on the network performance [12]. These researches mainly study the impact of traffic policing on traditional TCP. In contrast, we focus on studying the performance of the traffic policer with traditional and new CCs contending for bandwidth together.

Improving traffic policers. Similar to FairPolicer, DTB [52] aims to fairly allocate bandwidth among flows. DTB employs a token bucket for each active flow and each token bucket has the same fair share rate. However, DTB is non-work-conserving: some flows may send data at a rate lower than their fair share rate while other flows may desire more bandwidth than their fair share rate. Haalen et al. propose a scheme to dynamically adjust the token bucket size to optimize TCP's performance [11]. This work is orthogonal to FairPolicer. Similar to FairPolicer, CPQ [53] manages bandwidth sharing by classifying traffic on a per-customer basis and throttles traffic with per-customer traffic policers. However, CPQ contains a queue, which may inflate latency. Furthermore, CPQ incurs high overhead with a large number of customers.

Leveraging traffic policing for QoS. Traffic policer is an important component for providing QoS. Some work [54], [55] uses traffic policer to limit the traffic rate in wireless networks. Feng et al. explore the traffic policers for providing throughput guarantees [8]. Re-feedback [56] utilizes token bucket policers to equalize the flow rate. Furies [57] deploys a set of token bucket policers for malicious flow detection. These researches are complementary to ours. BwE [58] utilizes the token bucket algorithm to allocate bandwidth to different flow groups in Google's cloud network. Each flow group usually consists of multiple flows. BwE does not focus on how interior flows in each flow group share the allocated bandwidth. Thus, FairPolicer is complementary to BwE, which aims to enforce the allocated bandwidth to be fairly shared among different flows *inside* each flow group.

Fair Queuing. Traditionally, Fair Queuing [59], [60], [61], [62], [63], [64] is another method to ensure fair bandwidth allocation inside the network. However, Fair Queuing is difficult to be implemented with lots of concurrent flows. In contrast, FairPolicer can scale to a large number of flows at low cost. Recently, AFQ [39], EFQ [65], and HCSFQ [66] have shown that Fair Queuing can be implemented with programmable switches. AFQ and EFQ try to approximate Fair Queueing with limited FIFO queues. Compared with them, FairPolicer has two advantages when applied to the context of Internet traffic policing. (1) FairPolicer does not inflate latency with lots of flows. AFQ and EFQ allow different flows to be placed in the same queue. With lots of flows, the length of each FIFO queue can be very large and packets can experience high queueing latency. In comparison, FairPolicer does not contain a queue and thus does not introduce queueing latency. (2) FairPolicer requires less memory. AFQ and EFQ require memory to buffer packets, and each flow requires at least one packet for fair bandwidth sharing. With thousands of concurrent flows, AFQ and EFQ require tens-of-megabytes memory. In comparison, FairPolicer only requires 32KB memory to scale to thousands of flows.

HCSFQ is a hierarchical fair queueing scheduler designed for datacenter networks. Compared with it, FairPolicer has three advantages in the context of Internet traffic policing. (1) FairPolicer does not need to modify packet headers. HCSFQ requires all packets to carry the flow's arrival rate and hierarchy structure in the packet header. However, existing network devices on the Internet may not be able to recognize the packets with customized packet headers. (2) FairPolicer can work solely on network devices without any support from end/edge devices. HCSFQ requires the edge nodes to maintain the flow's traffic rate and inform network devices of the rate. FairPolicer does not make any assumptions about end/edge devices and thus can be more readily deployed. (3) FairPolicer does not require complex operations that are not easy to be realized on hardware. HCSFQ requires floating-point multiplications, divisions, and exponentiation operations, which can only be approximated on programmable switches. In comparison, FairPolicer only needs simple operations (e.g., comparison, addition, enqueue, dequeue). Thus, it is easier to be realized, especially when ISPs need to deploy thousands of traffic policers on a single device. In sum, HCSFQ is only applicable to a controlled environment (e.g., datacenter networks) and is not suitable for Internet traffic policing. Furthermore, FairPolicer can also bring some benefits (i.e., simplifying the implementation and deployment) in the controlled environments.

TABLE IV
KEY NOTATIONS

Not.	Description
B	Bucket capacity / Maximum burst size
R	Throttling rate
T_{token}	Token threshold
T_{ecn}	ECN threshold
$\theta^{(i)}(t)$	Residual bucket space for flow i at time t
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

VII. 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 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.

APPENDIX

A. Poof of Theorem 2

We start from a simple scenario that only one flow is passing through FairPolicer. In the scenario, we first derive the lower bound of T_{token} given T_{ecn} such that the flow can avoid packet drops. Then we drive the lower bound of T_{ecn} such that the flow can fully utilize the allocated bandwidth. Finally, we extend our analysis with n flows. The key notations are summarized in Table IV for the sake of terseness.

1) Lower Bound of T_{token} With 1 Flow: We start by deriving the lower bound of per-flow bucket capacity T_{token}, which is a prerequisite to derive the lower bound of B.

Given T_{ecn}, the bucket capacity of a flow should be large enough to avoid packet drops. Specifically, when packets for a flow are marked with T_{ecn}, it stills takes $\sim$1 RTT to feed back the ECN signal. In other words, when the residual bucket space reaches T_{ecn}, the flow will be consuming tokens for 1 RTT. To avoid packet drops, T_{token} should be large enough so that this flow won't exhaust tokens. In this paper, we call $T_{token} - T_{ecn}$ as *cushion tokens*.

A flow needs more cushion tokens in the slow start phase than the congestion avoidance phase, as the increasing rate of the congestion window is much higher in the slow start phase. Therefore, we only consider the slow start phase in this part. Besides, we assume that the senders do not pace packets, and the available bandwidth from each sender to FairPolicer is infinite. We make these unrealistic assumptions to analyze parameter settings in the worst case that traffic is extremely bursty. Furthermore, we assume that the available bandwidth from FairPolicer to each receiver is no lower than R.

We start from a simple scenario that only one flow is passing through FairPolicer. The bucket capacity of this flow is $T_{token} = B/2$. Let W_0 denote the initial congestion window and D denote the round-trip time. In the slow start phase, the congestion window at the k-th RTT can be given by $W_0 \cdot 2^k$. Let $\theta(t)$ denote the residual bucket space of the flow at time t.

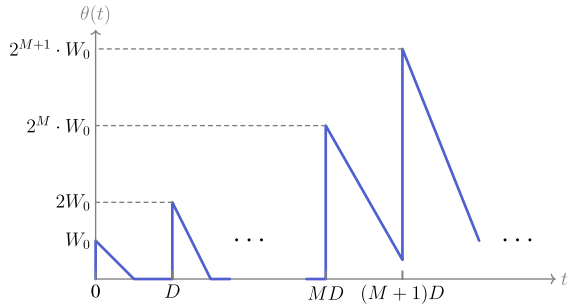


Fig. 15. Window evolution of CUBIC in the steady state.

Fig. 15 depicts the evolution of $\theta(t)$. As we have assumed that the bandwidth from senders to FairPolicer is infinite, the k-th round of packets will arrive at the policer all at once at time $t = kD$, occupying $W_0 \cdot 2^k$ tokens. During $[(k-1)D, kD]$, the policer will generate tokens at a rate of R, reducing the residual bucket space. At the beginning, the number of arriving packets is small and the number of generated tokens during two successive packet arrivals is larger than the number of used tokens. The bucket will be filled with tokens (i.e., the residual bucket space will become empty) before the arrival of packets. As the congestion window increases, the number of generated tokens is not enough to fill the bucket. Formally, we have

$$\theta(kD^+) = \begin{cases} W_0 \cdot 2^k, & k \leqslant M, \\ \theta((k-1)D^+) - RD + W_0 \cdot 2^k, & k > M \end{cases} \tag{14}$$

where $M = \max\left\{\left\lceil \log_2 \frac{RD}{W_0} \right\rceil, 0\right\}$ and $\theta(kD^+)$ is the token occupancy just after the arrival of packets. When $k > M$, recursively expanding the above align yields

$$\theta(kD^+) = \begin{cases} W_0 \cdot 2^k, & k \leqslant M, \\ W_0 \cdot \left(2^{k+1} - 2^M\right) - (k-M)RD, & k > M \end{cases} \tag{15}$$

With the closed-form expression of $\theta(t)$, we can derive the lower bound of T_{token}. Assume that the packets are marked with ECN at $t = jD$. This implies

$$\begin{cases} \theta(jD^+) > T_{ecn} \\ \theta\left((j-1)D^+\right) \leqslant T_{ecn} \end{cases} \tag{16}$$

To avoid packet drops, the token occupancy at time $t = (j+1)D$ should not exceed the bucket capacity, namely

$$\theta\left((j+1)D\right) \leqslant T_{token} \tag{17}$$

There are four cases.

Case 1: $T_{ecn} < W_0 2^{M-1}$

In this case, $j < M$. Thus, inequality (16) and (17) can be rewritten by

$$\begin{cases} T_{ecn} < W_0 2^j \\ T_{ecn} \geqslant W_0 2^{j-1} \end{cases} \tag{18}$$

$$T_{token} \geqslant W_0 2^{j+1} \tag{19}$$

According to (18), $W_0 2^{j+1} \leqslant 4T_{ecn}$. Thus, to avoid packet drops, we should have

$$T_{token} \geqslant 4T_{ecn} \tag{20}$$

Case 2: $W_0 2^{M-1} \leqslant T_{ecn} < W_0 2^M$

In this case, $j = M$. Inequality (16) and (17) can be rewritten by

$$\begin{cases} T_{ecn} < W_0 2^M \\ T_{ecn} \geqslant W_0 2^{M-1} \end{cases} \tag{21}$$

$$T_{token} \geqslant 3W_0 \cdot 2^M - RD \tag{22}$$

According to (21), $3W_0 2^M - RD \leqslant 6T_{ecn} - RD$. Thus, to avoid packet drops, we should have

$$T_{token} \geqslant 6T_{ecn} - RD \tag{23}$$

Case 3: $W_0 2^M \leqslant T_{ecn} < 3W_0 2^M - RD$

In this case, $j = M + 1$. Inequality (16) and (17) can be rewritten by

$$\begin{cases} T_{ecn} < 3W_0 2^M - RD \\ T_{ecn} \geqslant W_0 2^M \end{cases} \tag{24}$$

$$T_{token} \geqslant 7W_0 2^M - 2RD \tag{25}$$

According to (24), $7W_0 2^M - 2RD \leqslant 7T_{ecn} - 2RD$. Thus, to avoid packet drops, we should have

$$T_{token} \geqslant 7T_{ecn} - 2RD \tag{26}$$

Case 4: $T_{ecn} \geqslant 3W_0 2^M - RD$

In this case, $j - 1 > M$. Inequality (16) and (17) can be rewritten by

$$\begin{cases} T_{ecn} < \theta((j-1)D^+) - RD + W_0 2^j \\ T_{ecn} \geqslant \theta((j-2)D^+) - RD + W_0 2^{j-1} \end{cases} \tag{27}$$

$$T_{token} \geqslant \theta(jD^+) - RD + W_0 2^{j+1} \tag{28}$$

According to (27),

$$\begin{aligned} &\theta(jD^+) - RD + W_0 2^{j+1} \\ &\quad = \theta((j-1)D^+) - 2RD + 3\ W_0 2^j \\ &\quad = \theta((j-2)D^+) + 7W_0 2^{j-1} - 3RD \\ &\quad \leqslant 7\ T_{ecn} + 4RD - 6\theta((j-2)D^+) \\ &\quad \leqslant 7\ T_{ecn} + 4RD - 6\ W_0 2^{j-2} \\ &\quad \leqslant 7\ T_{ecn} + 4RD - 6\ W_0 2^M \\ &\quad < 7\ T_{ecn} - 2RD \end{aligned} \tag{29}$$

Thus, to avoid packet drops, we should have

$$T_{token} < 7T_{ecn} - 2RD \tag{30}$$

In summary, the lower bound of T_{token} can be given by

$$T_{token} \geqslant \begin{cases} 4T_{ecn}, & T_{ecn} < W_0 2^{M-1} \\ 6T_{ecn} - RD, & W_0 2^{M-1} \leqslant T_{ecn} < W_0 2^M \\ 7T_{ecn} - 2RD, & T_{ecn} \geqslant W_0 2^M \end{cases} \tag{31}$$

2) Lower Bound of T_{ecn} With 1 Flow: T_{ecn} should be large enough to ensure that the throughput of a flow can fully utilize the allocated bandwidth [11].

We start from a simple scenario that only one flow is passing through FairPolicer, which has an ECN threshold of T_{ecn}. In the congestion avoidance phase, the congestion window (denoted by $w(t)$) of CUBIC is determined by the following function [31]:

$$w(t) = C(t-K)^3 + w_{\max} \tag{32}$$

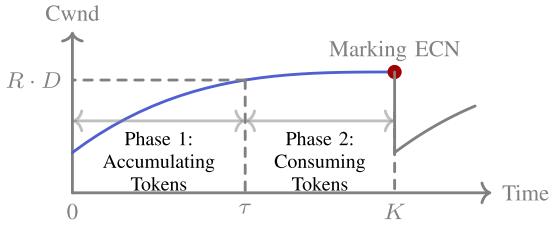


Fig. 16. Window evolution of CUBIC in the steady state.

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. 16 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 Phase 2 (i.e., $(\tau, K]$), the sending rate of the flow is higher than the throttling rate. Thus, the policer is consuming tokens from the bucket.

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 overflow in Phase 1. Formally, we should have

$$R\tau - \int_0^\tau \left[\frac{C(t-K)^3 + w_{\max}}{D}\right] \mathrm{d}t \leqslant T_{ecn} \tag{33}$$

To solve the above inequality, we need to solve for $w_{\max}$ and τ. As the number of tokens in the bucket at time 0 is the same as that in the time K, we have

$$\int_0^K \left[\frac{C(t-K)^3 + w_{\max}}{D}\right] \mathrm{d}t = R \cdot K \tag{34}$$

Solving (34) for $w_{\max}$ yields

$$w_{\max} = \frac{4}{4-\beta} RD \tag{35}$$

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 yields

$$\tau = \left[\sqrt[3]{\frac{4\beta}{C(4-\beta)}} - \sqrt[3]{\frac{\beta}{C(4-\beta)}}\right] \sqrt[3]{RD} \tag{36}$$

Plugging (35) and (36) into (33) yields

$$T_{ecn} \geqslant \frac{G}{D} \cdot (RD)^{\frac{4}{3}} \tag{37}$$

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

3) Parameter Settings of T_{ecn} and B With n Flows: With n active flows competing for bandwidth in FairPolicer, R should be replaced with R/n and T_{token} should be replaced with $B/(n+1)$. Thus, inequality (31) and (37) should be rewritten as

$$T_{token} \geqslant \begin{cases} 4T_{ecn}, & T_{ecn} < W_0 2^{M-1} \\ 6T_{ecn} - \frac{RD}{n}, & W_0 2^{M-1} \leqslant T_{ecn} < W_0 2^M \\ 7T_{ecn} - \frac{2RD}{n}, & T_{ecn} \geqslant W_0 2^M \end{cases} \tag{38}$$

$$T_{ecn} \geqslant \frac{G}{D} \cdot \left(\frac{RD}{n}\right)^{\frac{4}{3}} \tag{39}$$

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 *Proc. ACM Internet-QoE*, Aug. 2016, pp. 43–48.

[2] T. Flach et al., "An Internet-wide analysis of traffic policing," in *Proc. ACM SIGCOMM*, Aug. 2016, pp. 468–482.

[3] T. Flach, L. Pedrosa, E. Katz-Bassett, and R. Govindan, "A longitudinal analysis of traffic policing across the web," Dept. Comput. Sci., USC, Los Angeles, CA, USA, 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 *Proc. ACM SIGCOMM*, Aug. 2019, pp. 130–144.

[5] H. Guo and J. Heidemann, "Detecting ICMP rate limiting in the internet," in *Proc. PAM*, 2018, pp. 3–17.

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

[7] W.-C. Feng, D. D. Kandlur, D. Saha, and K. G. Shin, "Understanding TCP dynamics in an integrated services internet," in *Proc. IEEE NOSSDAV*, May 1997, pp. 279–290.

[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 Trans. Netw.*, 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 *Proc. IEEE LANMAN*, Mar. 2001, pp. 81–82.

[10] S. Sahu, P. Nain, C. Diot, V. Firoiu, and D. Towsley, "On achievable service differentiation with token bucket marking for TCP," in *Proc. ACM SIGMETRICS*, Jun. 2000, pp. 23–33.

[11] R. van Haalen and R. Malhotra, "Improving TCP performance with bufferless token bucket policing: A TCP friendly policer," in *Proc. IEEE LANMAN*, Jun. 2007, pp. 72–77.

[12] W. M. Zuberek and D. Strzeciwilk, "Modeling traffic shaping and traffic policing in packet-switched networks," *J. Comput. Sci. Appl.*, vol. 6, no. 2, pp. 75–81, Oct. 2018.

[13] K. Winstein and H. Balakrishnan, "TCP ex machina: Computer-generated congestion control," in *Proc. ACM SIGCOMM*, Aug. 2013, pp. 1–12.

[14] M. Dong, Q. Li, D. Zarchy, P. B. Godfrey, and M. Schapira, "PCC: Re-architecting congestion control for consistent high performance," in *Proc. USENIX NSDI*, 2015, pp. 395–408.

[15] Y. Zaki, T. Pötsch, J. Chen, L. Subramanian, and C. Görg, "Adaptive congestion control for unpredictable cellular networks," in *Proc. ACM SIGCOMM*, Aug. 2015, pp. 509–522.

[16] 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 *Proc. ACM MMSys*, May 2016, pp. 1–12.

[17] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson, "BBR: Congestion-based congestion control," *Queue*, vol. 14, no. 5, pp. 20–53, Oct. 2016.

[18] F. Y. Yan et al., "Pantheon: The training ground for internet congestion-control research," in *Proc. USENIX ATC*, 2018, pp. 731–743.

[19] M. Dong et al., "PCC Vivace: Online-learning congestion control," in *Proc. USENIX NSDI*, 2018, pp. 343–356.

[20] V. Arun and H. Balakrishnan, "Copa: Practical delay-based congestion control for the internet," in *Proc. USENIX NSDI*, Jul. 2018, pp. 329–342.

[21] S. Abbasloo, Y. Xu, and H. J. Chao, "C2TCP: A flexible cellular TCP to meet stringent delay requirements," *IEEE J. Sel. Areas Commun.*, vol. 37, no. 4, pp. 918–932, Apr. 2019.

[22] N. Jay, N. Rotman, B. Godfrey, M. Schapira, and A. Tamar, "A deep reinforcement learning perspective on internet congestion control," in *Proc. ICML*, 2019, pp. 3050–3059.

[23] T. Meng, N. R. Schiff, P. B. Godfrey, and M. Schapira, "PCC proteus: Scavenger transport and beyond," in *Proc. ACM SIGCOMM*, Jul. 2020, pp. 615–631.

[24] S. Abbasloo, C.-Y. Yen, and H. J. Chao, "Classic meets modern: A pragmatic learning-based congestion control for the internet," in *Proc. ACM SIGCOMM*, Jul. 2020, pp. 632–647.

[25] N. Garg. (Nov. 17, 2019). "Evaluating COPA congestion control for improved video performance." Facebook. [Online]. Available: https://engineering.fb.com/video-engineering/copa/

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

[27] A. Mishra, X. Sun, A. Jain, S. Pande, R. Joshi, and B. Leong, "The great internet TCP congestion control census," in *Proc. ACM SIGMETRICS*, Jun. 2020, pp. 59–60.

[28] 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 *Proc. USENIX NSDI*, 2012, pp. 253–266.

[29] M. Handley, "Delay is not an option: Low latency routing in space," in *Proc. ACM HotNets*, Nov. 2018, pp. 85–91.

[30] G. Cormode and S. Muthukrishnan, "An improved data stream summary: The count-min sketch and its applications," *J. Algorithms*, vol. 55, no. 1, pp. 58–75, Apr. 2005.

[31] S. Ha, I. Rhee, and L. Xu, "CUBIC: A new TCP-friendly high-speed TCP variant," *ACM SIGOPS Operating Syst. Rev.*, vol. 42, no. 5, pp. 64–74, Jul. 2008.

[32] Microsoft. (Nov. 2017). *Updates on Windows TCP*. [Online]. Available: https://datatracker.ietf.org/meeting/100/materials/slides-100-tcpm-updates-on-windows-tcp

[33] *Full Proof of Theorems of PCC Vivace*. Accessed: Jul. 31, 2020. [Online]. Available: http://ttmeng.net/pubs/vivace_proof.pdf

[34] M. Hock, R. Bless, and M. Zitterbart, "Experimental evaluation of BBR congestion control," in *Proc. IEEE ICNP*, Oct. 2017, pp. 1–10.

[35] S. Claypool, "Sharing but not caring—Performance of TCP BBR and TCP CUBIC at the network bottleneck," Worcester Polytech. Inst., Worcester, MA, USA, Tech. Rep. E-project-031819-113523, Mar. 2019.

[36] A. Langley et al., "The QUIC transport protocol: Design and internet-scale deployment," in *Proc. ACM SIGCOMM*, Aug. 2017, pp. 183–196.

[37] A. Kortebi, L. Muscariello, S. Oueslati, and J. Roberts, "Evaluating the number of active flows in a scheduler realizing fair statistical bandwidth sharing," in *Proc. ACM SIGMETRICS*, Jun. 2005, pp. 217–228.

[38] C. Hu, Y. Tang, X. Chen, and B. Liu, "Per-flow queueing by dynamic queue sharing," in *Proc. IEEE INFOCOM*, May 2007, pp. 1613–1621.

[39] N. K. Sharma, M. Liu, K. Atreya, and A. Krishnamurthy, "Approximating fair queueing on reconfigurable switches," in *Proc. USENIX NSDI*, 2018, pp. 1–16.

[40] 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 *Proc. USENIX NSDI*, 2017, pp. 67–82.

[41] A. K. Choudhury and E. L. Hahne, "Dynamic queue length thresholds for shared-memory packet switches," *IEEE/ACM Trans. Netw.*, vol. 6, no. 2, pp. 130–140, Apr. 1998.

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

[43] D. Shan, P. Zhang, W. Jiang, H. Li, and F. Ren, "Towards the fairness of traffic policer," in *Proc. IEEE INFOCOM*, May 2021, pp. 1–10.

[44] *Copa Code*. Accessed: Jun. 13, 2023. [Online]. Available: https://github.com/venkatarun95/genericCC

[45] *PCC Vivace Code*. Accessed: Jun. 13, 2023. [Online]. Available: https://github.com/PCCproject/PCC-Uspace

[46] R. K. Jain, D.-M. W. Chiu, and W. R. Hawe, "A quantitative measure of fairness and discrimination," Digit. Equip. Corp., Hudson, MA, USA, Res. Rep. TR-301, Sep. 1984.

[47] A. Rao, A. Legout, Y.-S. Lim, D. Towsley, C. Barakat, and W. Dabbous, "Network characteristics of video streaming traffic," in *Proc. ACM CoNEXT*, 2011, pp. 1–12.

[48] A. Botta, A. Dainotti, and A. Pescapé, "A tool for the generation of realistic network workload for emerging networking scenarios," *Comput. Netw.*, vol. 56, no. 15, pp. 3531–3547, Oct. 2012.

[49] V. Paxson and S. Floyd, "Wide area traffic: The failure of Poisson modeling," *IEEE/ACM Trans. Netw.*, vol. 3, no. 3, pp. 226–244, Jun. 1995.

[50] *ns-3*. Accessed: Jun. 13, 2023. [Online]. Available: https://www.nsnam.org/

[51] *Broadcom Tomahawk*. Accessed: Jun. 13, 2023. [Online]. Available: https://people.ucsc.edu/~warner/Bufs/tomahawk

[52] J. Kidambi, D. Ghosal, and B. Mukherjee, "Dynamic token bucket (DTB): A fair bandwidth allocation algorithm for high-speed networks," *J. High Speed Netw.*, vol. 9, no. 2, pp. 67–87, 2000.

[53] D. P. Wagner, "Congestion policing queues—A new approach to managing bandwidth sharing at bottlenecks," in *Proc. IEEE CNSM Workshop*, Nov. 2014, pp. 206–211.

[54] S. Ghazal and J. Ben-Othman, "Traffic policing based on token bucket mechanism for WiMAX networks," in *Proc. IEEE ICC*, May 2010, pp. 1–6.

[55] F. Yong Li and N. Stol, "QoS provisioning using traffic shaping and policing in 3rd-generation wireless networks," in *Proc. IEEE WCNC*, Mar. 2002, pp. 139–143.
[56] B. Briscoe, A. Jacquet, C. Di Cairano-Gilfedder, A. Salvatori, A. Soppera, and M. Koyabe, "Policing congestion response in an internetwork using re-feedback," in *Proc. ACM SIGCOMM*, Aug. 2005, pp. 277–288.
[57] C. Chuah, L. Subramarian, and R. Katz, "Furies: A scalable framework for traffic policing and admission control," Dept. Elect. Eng. Comput. Sci., UC Berkeley, Berkeley, CA, USA, Tech. Rep. UCB//CSD-01-1144, May 2001.
[58] A. Kumar et al., "BwE: Flexible, hierarchical bandwidth allocation for WAN distributed computing," in *Proc. ACM SIGCOMM*, Aug. 2015, pp. 1–14.
[59] J. B. Nagle, "On packet switches with infinite storage," in *Innovations in Internetworking*. Norwood, MA, USA: Artech House, 1988, pp. 136–139.
[60] A. Demers, S. Keshav, and S. Shenker, "Analysis and simulation of a fair queueing algorithm," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 19, no. 4, pp. 1–12, Aug. 1989.
[61] S. Keshav, "On the efficient implementation of fair queueing," *Internetw., Res. Exp.*, vol. 2, no. 3, pp. 157–173, 1991.
[62] P. E. McKenney, "Stochastic fairness queueing," in *Proc. IEEE INFOCOM*, Jun. 1990, pp. 733–740.
[63] M. Shreedhar and G. Varghese, "Efficient fair queueing using deficit round Robin," in *Proc. ACM SIGCOMM*, 1995, pp. 231–242.
[64] I. Stoica, S. Shenker, and H. Zhang, "Core-stateless fair queueing: Achieving approximately fair bandwidth allocations in high speed networks," in *Proc. ACM SIGCOMM*, 1998, pp. 118–130.
[65] J. Liu, J. Huang, N. Jiang, W. Li, and J. Wang, "Achieving high utilization for approximate fair queueing in data center," in *Proc. IEEE ICDCS*, Nov. 2020, pp. 932–942.
[66] Z. Yu, J. Wu, V. Braverman, I. Stoica, and X. Jin, "Twenty years after: Hierarchical core-stateless fair queueing," in *Proc. USENIX NSDI*, 2021, pp. 29–45.

Danfeng Shan (Member, IEEE) received the B.E. degree in computer science and technology from Xi'an Jiaotong University, China, in 2013, and the Ph.D. degree in computer science and technology from Tsinghua University, China, in 2018. He is currently an Associate Professor with the School of Computer Science and Technology, Xi'an Jiaotong University. His research interests include data center networks and congestion control.

Linbing Jiang received the B.E. degree in computer science and technology from Chongqing University, China, in 2020. He is currently pursuing the master's degree in computer science and technology with Xi'an Jiaotong University. His research interests include congestion control and programmable networks.

Peng Zhang received the Ph.D. degree in computer science from Tsinghua University in 2013. He was a Visiting Researcher with The Chinese University of Hong Kong and Yale University. He is currently a Professor with the School of Computer Science and Technology, Xi'an Jiaotong University, Xi'an, China. He is also with the MOE Key Laboratory for Intelligent Networks and Network Security. His research interests include verification, measurement, and security in computer networks.

Wanchun Jiang received the bachelor's and Ph.D. degrees in computer science and technology from Tsinghua University, China, in 2009 and 2014, respectively. He is currently an Associate Professor with the School of Computer Science and Engineering, Central South University, China. His research interests include congestion control, data center networks, and the application of control theory in computer networks.

Hao Li received the Ph.D. degree in computer science from Xi'an Jiaotong University in 2016. He is currently an Associate Professor with the School of Computer Science and Technology, Xi'an Jiaotong University. His main research interests include programmable networks and high-performance network functions.

Yazhe Tang received the Ph.D. degree in computer science from Xi'an Jiaotong University in 2002. He is currently an Associate Professor with the School of Computer Science and Technology, Xi'an Jiaotong University. His research interests include software-defined networking and network security.

Fengyuan Ren (Member, IEEE) received the B.A. and M.Sc. degrees in automatic control from Northwestern Polytechnic University, China, in 1993 and 1996, respectively, and the Ph.D. degree in computer science from Northwestern Polytechnic University in 1999. From 2000 to 2001, he worked as a Post-Doctoral Researcher with the Department of Electronic Engineering, Tsinghua University. In January 2002, he moved to the Department of Computer Science and Technology, Tsinghua University, Beijing, China, where he is currently a Professor. He coauthored more than 150 international journal and conference papers. His research interests include network traffic management, control in/over computer networks, wireless networks, and wireless sensor networks. He has served as a technical program committee member and the local arrangement chair for various IEEE and ACM international conferences.