Tuning the Aggressive TCP Behavior for Highly Concurrent HTTP Connections in Intra-datacenter

Tao Zhang, Jianxin Wang, *Senior Member, IEEE,* Jiawei Huang, *Member, IEEE,* Jianer Chen, *Senior Member, IEEE,* Yi Pan, *Senior Member, IEEE,* and Geyong Min, *Member, IEEE*

***Abstract*—Modern data centers host diverse HTTP-based services, which employ persistent TCP connections to send HTTP requests and responses. However, the ON/OFF pattern of HTTP traffic disturbs the increase of TCP congestion window, potentially triggering packet loss at the beginning of ON period. Furthermore, the transmission performance becomes worse due to severe congestion in the concurrent transfer of HTTP response. In this work, we provide the first extensive study to investigate the root cause of performance degradation of highly concurrent HTTP connections in DCN. We further present the design and implementation of TCP-TRIM, which employs probe packets to smooth the aggressive increase of congestion window in persistent TCP connection, and leverages congestion detection and control at end-host to limit the growth of switch queue length under highly concurrent TCP connections. The experimental results of at-scale simulations and real implementations demonstrate that TCP-TRIM reduces the completion time of HTTP response by up to 80%, while introducing little deployment overhead only at the end hosts.**

***Index Terms*—data center, HTTP, TCP**

I. Introduction

NOWADAYS, a significant number of online service providers employ the data centers to offer Internet-facing applications, such as web searching, accessing content, e-retailing, and advertisement [1], [2]. Since the application performance directly affects the enterprise revenue, the network operators try their best to shorten the service response time thus providing end-users with good experiences [3], [4]. For the consideration of equipment cost, however, network oversubscription is very common in existing infrastructures, which do not provide enough network capacity between the servers [5]. Thus, the network transfer becomes a bottleneck for the application performance. For example, in cluster computing applications like MapReduce and Dryad, data transfer accounts for more than 50% of job completion time [6].

However, due to its wide usage in past 20 years, Hyper Text Transfer Protocol (HTTP) is foundation of the Internet-facing applications in modern data center. For example, the well-known MapReduce framework adopts HTTP or HTTPS to fetch the relevant partition of output of all the mappers in the shuffle phase [8], [9], [10], [11], [12]. After receiving an HTTP request message from end-user, the Web server in data center, which provides resources such as HTML files and other content, or performs other functions on behalf of the client, returns HTTP response to the end-user [13]. To achieve fast feedback and high reliability, the Web server usually utilizes highly concurrent HTTP connections to fetch the response data across a large number of compute and storage servers [14]. Previous research has reported that, the HTTP-based application contributes to nearly 85% and 50% of traffic in data centers of private enterprise and university campus, respectively [1].

Naturally, HTTP employs TCP as its underlying transport-level protocol, and generally maintains persistent TCP connections on which requests and responses are allowed to multiplex to reduce the unnecessary overhead caused by frequent three-way handshakes (SYN and FIN) [15], [16]. However, there are two key factors that together impair the performance of highly concurrent HTTP connections on TCP flows in data centers.

First, the nature of HTTP request/response style, coupled with the unpredictable and uncontrollable user's behavior, shapes the ON/OFF traffic pattern on the persistent TCP connection [16], [17]. This pattern, however, disturbs the self-clocking mechanism of TCP's control loop. Specifically, ON/OFF HTTP traffic makes the data transfer on the persistent TCP connection become non-successive. When waiting for the user request or server response, the TCP connection becomes idle, but is kept alive. Once the connection restarts after the idle time, it begins transmission with the congestion window (CW) inherited from the previous ON period, resulting in the aggressive increase in sending rate and potential congestion.

Second, inside the data center, multiple servers and their unique invoker constitute the many-to-one communication pattern [14]. For example, in order to respond to a user request of web search, hundreds, even thousands of web and database servers are involved in the compute and communication process across the data center network (DCN) [3], [4]. Such many-to-one traffic patterns, joint with the droptail queue management of switch buffer, bring about frequent buffer overflow and packet losses. Furthermore, when incorrectly inheriting congestion window from the previous ON period, the concurrent TCP connections which transport the HTTP traffic get substantially worse performance.

The bursty behavior of TCP on HTTP connection has already been investigated in the WAN context [18], [19].

Manuscript received July 6, 2016.

Tao Zhang, Jianxin Wang, Jiawei Huang are with School of Information Science and Engineering, Central South University, Changsha, China, 410083. E-mail: jxwang@mail.csu.edu.cn.

Jianer Chen is with Department of Computer Science and Engineering, Texas A&M University, College Station, Texas 77843, USA. E-mail: chen@cs.tamu.edu.

Yi Pan is with Department of Computer Science, Georgia State University, Atlanta, GA 30302-4110, USA. E-mail: yipan@gsu.edu.

Geyong Min is with the College of Engineering, Mathematics and Physical Sciences, University of Exeter, Exeter, EX4 4QF, U.K. E-mail: G.Min@exeter.ac.uk.

However, due to the huge difference of physical environment, the existing schemes in WAN fail to resolve the bursty problem in DCN [3], [4], [20], [21]. Furthermore, although the existing data center network TCPs also have adopted miscellaneous schemes to alleviate congestion of concurrent TCP flows, when transferring HTTP traffic, they share the same feature: inheriting congestion window from the previous ON period. Based on our empirical results in Section II, the TCP protocol cannot cope with the situation of concurrent HTTP connections. Under such a situation, the switch buffer easily suffers from frequent overflows, resulting in TCP timeout and throughput collapse.

In this work, based on the typical ON/OFF HTTP workload, we empirically study the transmission performance of persistent TCP connections. We reveal that the TCP's aggressive behavior in increasing congestion window causes TCP timeout and throughput collapse for the highly concurrent HTTP traffic. This is because TCP blindly inherits the congestion window from the previous ON period, even the congestion state has significantly changed during the OFF period.

Specifically, to solve this problem, we design a novel window inheritance mechanism, in which the congestion window size in the pervious ON period is conditionally reused to control the risk of heavy congestion. To smooth the switch queue leap caused by the concurrent data transfers, our design also regulates the congestion window size according to the end-to-end delay. The contributions of this paper are as follows:

- We present the first extensive study to investigate the root cause of performance degradation of highly concurrent HTTP connections in DCN. We reveal the impact of HTTP's ON/OFF style on TCP protocol in the scenario of concurrent transfer, and demonstrate experimentally why the congestion window of the persistent TCP flow should be elaborately controlled at the beginning of ON period.
- We propose a new transport protocol, TCP-TRIM, which employs probe packets to detect congestion state and smooth the aggressive increase in congestion window at end hosts. By selectively inheriting the window size and maintaining the queue delay near a target value, TCP-TRIM avoids the TCP timeout and throughput collapse. Based on the theoretical analysis of steady state behavior, we also give a guideline for choosing the threshold in order to reduce congestion window in TCP-TRIM.
- We design an asynchronous timer (AT) as a component of TCP-TRIM to handle the high concurrency scenario. Through actively regulating the start time of new packet train, AT helps TCP-TRIM support the much higher concurrent data transfers in typical Partition/Aggregation communication pattern.
- We evaluate the performance of TCP-TRIM by using at-scale NS2 [22] simulations, and also on a small-scale testbed. The results show that, under highly concurrent HTTP connections, TCP-TRIM effectively avoids the TCP timeout and yields remarkable performance gain (i.e., up to 80%) in reducing the average completion time of HTTP response.

The remainder of this paper is organized as follows. The design motivation of TCP-TRIM is presented in Section II. In Section III, we describe the details of TCP-TRIM and present the model analysis. We evaluate the performance of TCP-TRIM on NS2 and real testbed in Section IV. The related works are discussed in Section V. Finally, we make conclusion in Section VI.

II. Motivation

In this section, we present empirical studies to demonstrate the ON/OFF pattern of HTTP traffic, and show it is very common in the modern data centers with highly concurrent HTTP connections. Then, we analyze why the TCP protocol fails to provide satisfactory performance. Finally, we present the design objectives.

A. ON/OFF Pattern in HTTP Traffic

To provide high-quality HTTP service, data center hosts plenty of servers that play different roles in generating objects, such as images, news, videos, and advertisements. When the end users send their requests to an assigned server (i.e., front-end server) [15], it parses the requests and invokes the back-end servers to generate the responses. Then, these responses are returned to the front-end server and finally shown in front of the end user [3], [14].

For comprehensively understanding the characteristic of HTTP traffic, we recorded the real workload of data center at Central South University, China. Processing the 2 TB trace data, we find that the HTTP traffic presents the ON/OFF pattern, which is exactly consistent with the statements in [1]. We also analyze a packet trace of a commodity data center from [7]. This trace involves more than 300,000 TCP connections, and about half of them have long packet inter-arrival time. Moreover, the maximum packet inter-arrival time of some connections even reaches to about 5 minutes, showing that plenty of connections have experienced OFF periods.

To describe the ON/OFF pattern more clearly, we define a packet train (PT) as a burst of packets on an HTTP connection from the identical source to the identical destination. If the time interval between two packets exceeds an inter-train gap, they belong to different trains [23].

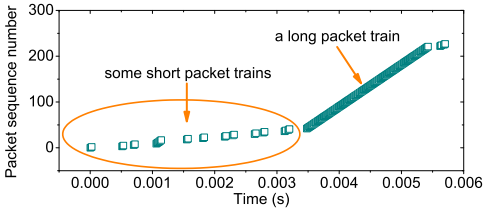
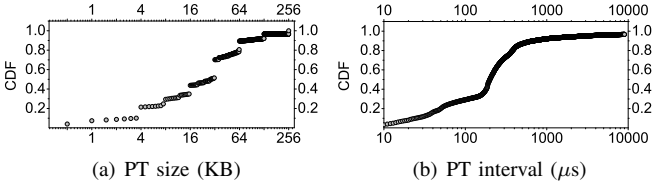


Fig. 1. Understanding the "Packet Train"



(a) PT size (KB) (b) PT interval (μs)

Fig. 2. CDFs of PT size and interval

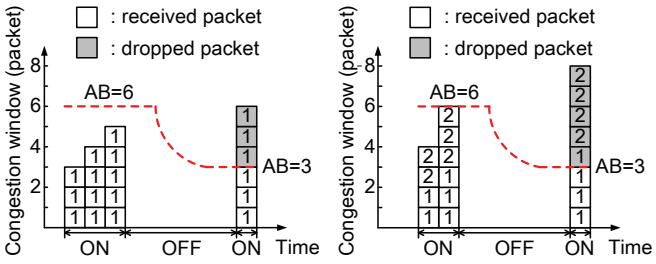
We trace the HTTP traffic data generated by a selected web server and plot the packet sequence number in Fig. 1. It is shown that different sizes of packet trains are sent

by the web server intermittently. Since the packets of long packet train (LPT) are transferred almost in a stream way, they contribute much more to the increase of packet sequence number compared to the packets of short packet train (SPT). On the contrary, SPT often shows the behaviors of burst and intermittence. Moreover, the number of packets in each SPT varies from a few to dozens, while LPT carries nearly one hundred packets or more. We also measure the data size and inter-train gap of all PTs in the traffic trace. As shown in Fig. 2(a), the data size of PT varies from 0.5 KB to 256 KB, and about 70% is between 4 KB to 128 KB. The proportion of tiny PTs (i.e., $\leqslant$ 4 KB) is lower than 20%, while 10% is larger than 128 KB. However, as shown in Fig. 2(b), the inter-train gap lasts from hundreds of microseconds to several milliseconds.

B. Performance Impairments

HTTP connection builds up persistent TCP flow to reduce the overhead from frequent three-way handshakes. However, this delicate configuration combined with the concurrent data transfer potentially impairs the transmission performance. The detail description is given as follows.

1) Congestion Window of Persistent TCP Connection: Considering the frequent request/response interactions of HTTP, if it has to build a new TCP connection for each response, the massive operation for connection setup and teardown will waste the network bandwidth and system resources. Thus, the prevalent versions, such as HTTP 1.0 and 1.1, all build up single persistent TCP flow and enable multiple requests and responses to share such a single flow [16]. Although the transfer efficiency is improved, there also comes another issue: since in HTTP each PT starts with the window size inherited from the previous PT, once a PT with plenty of packets arrives with inheriting a large congestion window, massive packets will be instantaneously injected into the bottleneck link thus potentially generating heavy congestion.

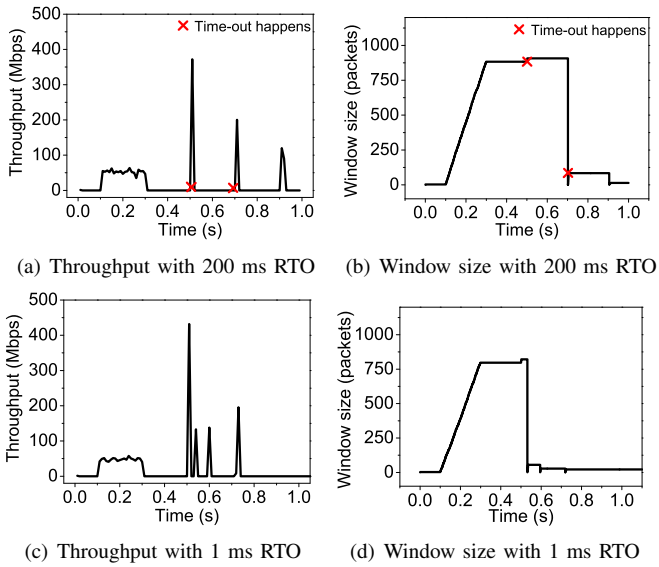


(a) Only connection 1 (b) Two concurrent connections

Fig. 3. Transferring PTs on persistent TCP connections.

This issue is visually described in Fig. 3. Wherein "ON" means HTTP connection is active, while there is no traffic during OFF period. AB (measured in packets) indicates the current available bandwidth. If the total number of flying packets is larger than AB, packet loss will happen. Fig. 3(a) shows the window evolution of connection 1 on the bottleneck link. During the first ON period of connection 1, no congestion happens because its congestion window, w_1, is never more than AB. Thus, at the end of the first ON period, w_1 expands from 5 to 6 and is maintained until the second ON period starts. However, during the OFF period of connection 1, some incoming traffic from other connections takes up some available bandwidth so that AB decreases to 3. Unfortunately, this situation is not perceived by connection 1, thereby at the beginning of its second ON period, 3 packets are dropped. When it comes to the case of multiple concurrent connections, this impairment becomes much severer. As shown in Fig. 3(b), two coexisting connections cause all the packets in the congestion window of connection 2 to be dropped, resulting in TCP timeout [24].

For further elaborating this impairment, we install the synthetic traffic derived from our real trace data on a many-to-one scenario built on NS2. Specifically, five servers connect to a front-end server via a switch with 100 packet buffer through five 1 Gbps links with 50 μs latency. The front-end server sends requests to each server. Each server immediately returns a response after receiving a request. This request/response style is very common in the HTTP applications. In this scenario, each server totally returns 200 responses from 0.1s sequentially. The data size of each response ranges from 2 KB to 10 KB, and the interval between two neighboring responses is randomly generated based on the mean 1 ms. After that, each server sends a LPT with more than 128 KB data at 0.5 s. All the connections run on TCP Reno, and packet size is set as 1460 bytes. The retransmission timeout (RTO) is 200 milliseconds as default. Meanwhile, we keep the 5 TCP connections throughout the whole transmission.



(a) Throughput with 200 ms RTO (b) Window size with 200 ms RTO

(c) Throughput with 1 ms RTO (d) Window size with 1 ms RTO

Fig. 4. Throughput and window size of connection 5 with different RTO.

We trace the test results and find that most of the connections involve the occurrence of TCP timeouts. To be specific, except connection 1, the numbers of timeouts in connections 2, 3, 4, and 5 are 1, 2, 2, and 2, respectively. For simplicity, we just select connection 5 to make a concrete analysis of throughput and congestion window. From Fig. 4(a), we observe that two TCP timeouts happen at about 0.5 s and 0.7 s, hence the network efficiency is greatly degraded. To clarify whether the blind window inheritance is the culprit of performance degradation, we plot the window evolution of

connection 5 in Fig. 4(b). Wherein the window size is close to 900 at 0.3 s, and kept until 0.5 s, which is the start time of LPT transmission. Meanwhile, the inherited window sizes in connections 1, 2, 3, and 4 all exceed 850 packets. Obviously, such huge windows bring heavy congestion to the bottleneck link, where the allowed number of flight packets is at most 118 (the summation of Bandwidth Delay Product and switch buffer size) in this scenario. Besides, we also use a small RTO (1 ms) to run the simulation again, while the problem still exists, and the window size is unacceptable yet, as shown in Fig. 4(c) and Fig. 4(d).

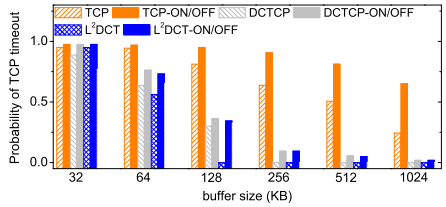
In essence, the data size of each response is too small to generate packet losses, so the sender mistakenly believes that the current congestion window is still small and will not bring about congestion, thus continuously expanding its congestion window. Once LPT arrives, the sender spontaneously inherits the congestion window in the previous ON period and sends as many packets as possible in one RTT, thereby inducing heavy congestion. In general, TCP sender immediately sends a new packet once receiving a desired ACK, however, if this consecutive process is broken by HTTP ON/OFF pattern, there is no reason that the sender can still directly send data based on the congestion window in the previous ON period.

Although some studies on WAN have already investigated the problem caused by the ON/OFF traffic, their schemes may not work because the physical environment of DCN is very different from that of WAN. As described in [18], the existing idle detection schemes in WAN cannot perceive an idle period shorter than an RTO, while in DCN plenty of idle periods are far from reaching the level of RTO (see Fig2(b)). Moreover, the proposed methods for controlling the packet bursts lack accurate bandwidth detection, thus fail to guarantee both high bandwidth and low delay transmission, which is exactly the crucial requirement for designing the data center transport protocol [20], [21].

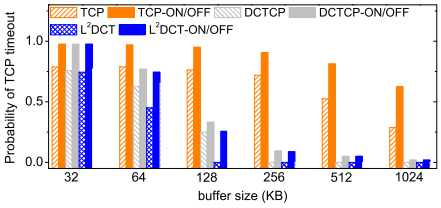
2) Impairments on Concurrent HTTP Connections: In commodity data centers, the communication pattern of Partition/Aggregation is prevalent and plays an important role for providing HTTP-based services. In this pattern, a user request is first distributed to hundreds, even thousands of servers to calculate responses, called Partition. Then these responses are sent back to the aggregation servers at nearly the same time, which is the Aggregation [3]. In the aggregation phase, the response traffic is unpredictable and often bursts in many-to-one way, potentially leading to abundant packet losses and TCP timeouts.

A similar problem that also happens in the scenario of concurrent TCP transfer, namely TCP Incast, has been widely investigated, and most of the works deal with it from modifying the default settings of TCP [25], [26], [27], preventing the switch queue buildup [3], and desynchronizing the transfer [28], etc. However, to the best of our knowledge, none of them specifically focuses on the impact of ON/OFF traffic on concurrent HTTP connections [20], [21]. In the following part, we show how concurrent HTTP connections further impair the TCP transmission performance in the many-to-one scenario.

We rebuild the previous many-to-one scenario, and increase the sender number from 2 to 40 step by step with varying link bandwidth and latency ([1Gbps, 50 μs] and [10Gbps, 25 μs], respectively). In each case of the number of senders, we vary the data size of LPT from 32 KB to 256 KB with gradually expanding the switch buffer (32 KB to 1 MB), and carry out the "many-to-one" data transfers for a certain number of times (about 1000 times) to observe if each run has experienced the TCP timeout. Moreover, our tests are run based on two situations, with and without the impact of ON/OFF traffic, respectively. Besides, except the vanilla TCP, also we observe if the state-of-the-art data center transport protocols, such as DCTCP and L^{2}DCT, suffer from a severer performance degradation after introducing the ON/OFF HTTP traffic.



(a) 1Gbps, 50 μs



(b) 10Gbps, 25 μs

Fig. 5. More severer performance degradation.

In Fig. 5, with the impact of ON/OFF traffic, enlarging the buffer for vanilla TCP does not greatly inhibit the occurrence of TCP timeout. The probability of TCP timeout still remains at a high level (more than 60% even with 1 MB buffer size). Although other protocols perform a little better, the risk of TCP timeout still exists after introducing the ON/OFF HTTP traffic.

C. Summary

The above observation leads us to conclude that (i) the ON/OFF pattern of HTTP traffic disturbs the increasing of TCP's congestion window, potentially triggering packet loss at the beginning of ON period, and (ii) the transmission performance becomes worse due to severe congestion in the highly concurrent data transfer. These conclusions motivate us to investigate a novel approach smoothing the aggressive increasing of congestion window on persistent TCP connection. In the rest of this paper, we present our TCP-TRIM as well as a reference implementation in real testbed system.

III. TCP-TRIM

In this section, we firstly describe the design detail of TCP-TRIM. Then, based on the theoretical analysis of the steady state behavior, we give a guideline for determining the threshold that is used for reducing congestion window in TCP-TRIM. Besides, we present the details of asynchronous time, which is used to help TCP-TRIM deal with highly concurrent data transfer.

A. Design Details

The design goal of TCP-TRIM is to tune the aggressive TCP behavior for highly concurrent HTTP connections in data center. To achieve this goal, however, TCP-TRIM faces four key challenges that (i) TCP-TRIM should obtain the accurate congestion level when HTTP enters into the ON period, (ii) TCP-TRIM should smooth the expansion of the congestion window, while ensuring high utilization of bottleneck link, (iii) TCP-TRIM should be able to cope with the highly concurrent HTTP data transfer, and (iv) be easy to be deployed without hardware refresh on switch, because the current trend is to use cheap, Commercial Off-the-Shelf (COTS) switches to construct large-scale data center networks [27]. In the following, we describe the design details of TCP-TRIM.

1) Detecting inter-train gap: When packet loss does not happen, the arrival of an ACK immediately triggers the sending of next data packet. Hence the time interval between any two neighboring packets in a PT is supposed to be less than the round trip time. Based on this consideration, TCP-TRIM needs to sense RTT for each packet, and considers the smoothed RTT, which is calculated in Algorithm 2, as the inter-train gap. As described in Algorithm 1, before sending a new packet, the TCP-TRIM sender calculates the time interval ti between the current time and the sending time of last packet. If ti is larger than the smoothed RTT, the sender begins to detect congestion and smooth the sending rate. Specifically, the sender records the current size of congestion window $cwnd$ and sets it to 2. Then, only two packets are sent out in the current window and are used as probe packets. Meanwhile, the sender pauses the data transfer, waiting for ACKs of the two packets.

Algorithm 1 : Inter-train gap detection

```
Before sending a new packet :
( not a retransmission packet )
if ti > smooth_RTT then
    Saving the accumulated window size;
    cwnd ← 2;
    Sending the probe packets in current window;
    Suspending the packet transfer;
end if
Call Algorithm 2;
```

Note that we do not claim that our method can identify whether a packet that will be sent belongs to a new PT. In effect, TCP-TRIM determines if the probe packets should be sent from the viewpoint of packet level. The reason is that larger RTT may occur between two neighboring packets that belong to one PT. However, if ti exceeds the smoothed RTT during one packet train's transfer, it indicates that the connection is experiencing congestion. Under this situation, it is still necessary to detect congestion and smooth sending rate.

2) Tuning congestion window: For each arriving ACK, TCP-TRIM measures the current RTT, and updates three variables in following three operations: (i) updating min_RTT, which is the link latency without switch queuing, (ii) determining the RTT threshold K based on min_RTT, and (iii) calculating $smooth\_RTT$, which is a smooth value of the current RTT. These variables are kept by the TCP connection hosted by the sender. TCP-TRIM works as the following two cases, which is shown in Algorithm 2.

Algorithm 2 : Ack action

```
For each arriving ACK ( not a duplicated ACK ) :
smooth_RTT ← (1 − α)smooth_RTT + αRTT;
if RTT < min_RTT then
    min_RTT ← RTT;
    Update K;
end if
if the current Ack belongs to the probe packet then
    if it arrived in a smooth_RTT then
        cwnd ← s_cwnd(1 − (probe_RTT − min_RTT)/min_RTT);
        Resume packet transfer;
    else
        cwnd ← 2;
        Resume packet transfer;
    end if
else
    if RTT ⩾ K then
        ep ← (RTT − K)/RTT;
        cwnd ← cwnd(1 − ep/2);
    else
        cwnd ← cwnd + 1/cwnd;
    end if
end if
```

If the current Ack belongs to the probe packet, the sender begins to smooth the increasing of congestion window. If any ACK of probe packet does not come back in a smoothed RTT, the sender immediately sets the window size to 2, which is the default value of minimum congestion window in TCP. Otherwise, the sender tunes the current window size by

$$cwnd = s\_cwnd(1 - \frac{probe\_RTT - min\_RTT}{min\_RTT}), \quad (1)$$

where s_cwnd is the saved window size and $probe\_RTT$ is the average value of the probe packets. After this operation, the sender restarts the transfer of remained packets based on the tuned window size. Some previous works, such as [24], just send the new PT with congestion window size 2 to minimize the congestion possibility. However, this conservative method may underutilize the bottleneck link if the network has enough capacity to accommodate a large window size.

If the arriving ACK does not belong to the probe packets, the sender enters the queuing control phase. TCP-TRIM measures the current RTT to monitor the real-time congestion level. In our design, when the RTT exceeds the predefined threshold K, it is convinced that packets have been buffered in the switch queue. The proportion (denoted by ep) of the exceeded part to the current RTT represents the congestion level, which is calculated by

$$ep = \frac{RTT - K}{RTT}. \quad (2)$$

Then we can also approximately deduce that in the current congestion window there are $ep \times cwnd$ packets that should not

be ejected into the bottleneck link. However, in the high speed DCN where only a small number of flows share the switch buffer [3], directly using $(1-ep)\times cwnd$ to shrink window may cause a large mismatch between the input rate to the link and the available capacity, resulting in buffer underflows and loss of throughput. Based on this consideration, we borrow the idea from DCTCP [3] and stipulate that the window reduction of TCP-TRIM can not be more aggressive than that of the legacy TCP. Hence, when the sender finds its RTT is larger than a predefined threshold K, its congestion window is adjusted to

$$cwnd = cwnd(1 - \frac{ep}{2}). \tag{3}$$

Note that the Slow-Start, additive increase in Congestion Avoidance, Fast Retransmit, and Fast Recovery of the conventional TCP remain unchanged in TCP-TRIM.

3) Guideline for choosing K: generally, the traffic on the HTTP connection can be divided into two states, the ON/OFF state and the successive ON state. TCP-TRIM employs its "inter-train gap detection" mechanism to deal with the ON/OFF traffic. As to the successive traffic, TCP-TRIM turns to control the queue buildup at the switch by sensing RTT in real time. Once the measured RTT is greater than K, the TCP-TRIM sender promptly carries on its backoff operation. Hence, in this part, we analyze how to choose an appropriate K to deal with the successive traffic for achieving high utilization, low latency, and transferring data without TCP timeout as well. Our method is based on the approaches reported in [3] and [29]. Next, we respectively discuss the lower and upper bound of K through analyzing the steady state behavior of TCP-TRIM.

The Lower Bound (LB) of K: Intuitively, if K is set as a smaller value, the congestion window is decreased in advance thus avoiding the congestion earlier. However, this may cause the insufficient use of bottleneck link capacity when K is too small. Next, we discuss how to choose an appropriate K guaranteeing the high bottleneck link utilization.

Suppose that N persistent TCP connections are totally synchronized, and maintained between N servers and a single front-end server. Each web server sends a single LPT with infinite packets via the bottleneck link with capacity C (in packets per second). The round trip time without queueing between a server and the front-end host is D (measured in seconds), and K is the RTT threshold for window back-off, thus $K-D$ represents the allowed queueing latency, then we get the desired switch queue length Q by

$$Q = C(K - D). \tag{4}$$

In the meantime, the number of packets that can be allowed to stay in the network is CK, and for each synchronized PT the allowed maximum value of window size is CK/N.

Assume that at time t, the queue length of switch is just equal to Q, and at the same time each PT is in the ith RTT, then we get the window size of each PT in the ith RTT by

$$W_{(i)} = \frac{CK}{N}. \tag{5}$$

Since $W_{(i)}$ does not result in the switch queue length that exceeds Q, no PT will backoff and the window size of each PT in the $(i+1)$st RTT is

$$W_{(i+1)} = \frac{CK}{N} + 1. \tag{6}$$

However, the window size of each PT in the $(i+1)$st RTT will make the queue length exceed Q, hence each connection will slow down, and then the maximum queue length Q_{max} is

$$Q_{max} = C(K - D) + N. \tag{7}$$

In fact, Eq. (7) also indicates that each of N packets belongs to a corresponding PT, queues one by one behind the allowed queue part, hence we calculate the current RTT by

$$RTT_{(i+1)(j)} = K + \frac{j}{C}, \tag{8}$$

wherein j represents the jth PT, and $j = 1, 2, 3, ..., N$. From Eqs. (2) and (3), we also get the current congestion level by

$$ep_{(i+1)(j)} = \frac{j}{CK + j}. \tag{9}$$

Thus the total sum of window decrement $\Delta cwnd_{(i+1)(j)}$ for all the PTs before the $(i+2)$nd round's transfer is

$$\sum_{j=1}^{N} \Delta cwnd_{(i+1)(j)} = \left(\frac{CK + N}{2N}\right) \sum_{j=1}^{N} \frac{j}{CK + j}. \tag{10}$$

For guaranteeing the 100% utilization of bottleneck link, the switch queue length should never be less than 0, then we have

$$Q_{max} - \sum_{j=1}^{N} \Delta cwnd_{(i+1)(j)} > 0. \tag{11}$$

By substituting Eq. (7) into Eq. (11), we get

$$C\,(K - D) + N - \left(\frac{CK + N}{2N}\right) \sum_{j=1}^{N} \frac{j}{CK + j} > 0. \tag{12}$$

wherein $\sum_{j=1}^{N} j/(CK + j)$ could be approximately considered as

$$\int_{1}^{N} \frac{j}{CK + j} d_j = N - 1 + CK \ln \frac{CK + 1}{CK + N}. \tag{13}$$

With Eqs. (13) and (12), we get

$$C\,(K - D) + N > \left(\frac{CK + N}{2N}\right) \left(N - 1 + CK \ln \frac{CK + 1}{CK + N}\right). \tag{14}$$

Since $\ln(CK+1)/(CK+N) < 0$, Eq. (13) is smaller than $N-1$. Then we just need to let

$$C\,(K - D) + N > \left(\frac{CK + N}{2N}\right)(N - 1). \tag{15}$$

Meanwhile, we simplify it and get

$$K > \frac{2ND}{N+1} - \frac{N}{C}. \tag{16}$$

By analyzing the right part of Eq. (16), we could create a function about N ($N > 0$) by

$$F(N) = \frac{2ND}{N+1} - \frac{N}{C}, \tag{17}$$

wherein D and C are two constants, and N is the independent variable. It is intuitively plausible that $2ND/(N+1)$ has the function limit $2D$ and the limit of N/C is $+\infty$, hence $F(N)$ should have a upper bounder and be a Convex function. Then we get

$$\frac{dF(N)}{dN} = \frac{2D - \frac{N^2}{C} - \frac{2N}{C} - \frac{1}{C}}{(N+1)^2}. \tag{18}$$

To judge whether $F(N)$ has a stationary point we also get

$$\frac{N^2}{C} + \frac{2N}{C} + \frac{1}{C} - 2D = 0. \tag{19}$$

Since $8D/C > 0$, Eq. (19) has a positive solution and $F(N)$ has a stationary point. Next, the second derivative of $F(N)$ can be represented by

$$\frac{d(dF(N))}{dN} = \frac{-4D}{(N+1)^3}. \tag{20}$$

Thereby $F(N)$ has a maximum value for the reason that Eq. (20) is less than 0. Therefore, through solving Eq. (19), we obtain the maximum value of $F(N)$ and get

$$F(N) \leqslant \frac{\left(\sqrt{2CD} - 1\right)^2}{C}. \tag{21}$$

Clearly, if 100% bottleneck link utilization is supposed to be guaranteed at any time, K should be higher than $F(N)$. Meanwhile, K should also be larger than or equal to D. Therefore, these limitations lead us to determine the lower bound of K by

$$K \geqslant \max\left(\frac{\left(\sqrt{2CD} - 1\right)^2}{C}, D\right). \tag{22}$$

The Upper Bound (UB) of K: Once TCP timeout happens, the maintained TCP connection becomes idle so that TCP throughput collapses, leading to the sharp decline in transmission performance. Since a very large K potentially generates packet losses and causes TCP timeouts, we analyze the upper bound of K that provides efficient transfer without TCP timeout. In the TCP control loop, the window expansion in slow-start phase is more aggressive than that in congestion avoidance phase thus more likely generating packet losses, hence we focus on the slow-start phase to find out the upper bound of K.

Consider an extreme situation where N persistent TCP connections are totally synchronized, and in the slow-start phase for the data transfer in the ith RTT. Suppose that their packets in the ith RTT just make the switch queue length grow at CK. Then we have

$$\sum_{j=1}^{N} W_{(i)(j)} = CK. \tag{23}$$

Since the RTT of each connection is not greater than K, they all stay in the slow-start phase, and their total number of packets in the $(i+1)$st RTT is $2CK$. To avoid the packet loss in the $(i+1)$st RTT, we just need

$$2CK \leq B + CD, \tag{24}$$

wherein B (measured by packets) is the switch buffer size. Through solving Eq. (24), we get the upper bound of K by

$$K \leq \frac{B + CD}{2C}. \tag{25}$$

Discussion: To guarantee both high bottleneck link utilization and avoid TCP timeout, from Eqs. (25) and (22) we can choose K by

$$K \in \left[\max\left(\frac{\left(\sqrt{2CD} - 1\right)^2}{C}, D\right), \frac{B + CD}{2C}\right]. \tag{26}$$

Nonetheless, Eq. (26) needs

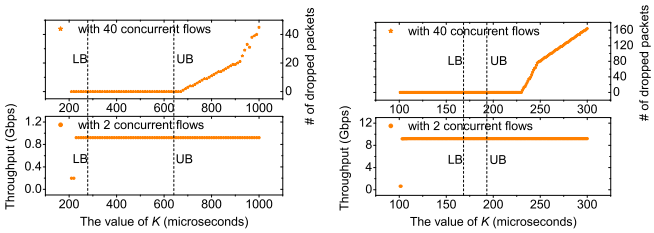
$$\frac{B + CD}{2C} \geqslant \max\left(\frac{\left(\sqrt{2CD} - 1\right)^2}{C}, D\right). \tag{27}$$

That is to say, when the lower bound of K is $\left(\sqrt{2CD} - 1\right)^2/C$, the switch buffer size should satisfy

$$B \geqslant 3CD - 4\sqrt{2CD} + 2. \tag{28}$$

At the same time, if the lower bound of K is D, the buffer size should be

$$B \geqslant CD. \tag{29}$$



(a) 1Gbps, 50 μs (b) 10Gbps, 25 μs

Fig. 6. Testing K with different link capacity and latency.

Next, we run NS2 experiments by assigning different link parameters ([1Gbps, 50 μs] and [10Gbps, 25 μs], respectively) to test if the threshold K determined from Eq. (26) works well. To be specific, each sender server sends a long TCP flow to a single receiver via a switch, and we gradually increase the number of senders from 2 to 40 so that the many-to-one scenario scales up. In each case of sender number, we evenly select many sample values of K from the interval of Eq. (26), and run test to observe the throughput and the number of dropped packets.

Intuitively, the bottleneck link is prone to be underutilized when there are only two senders, and it is more likely to generate packet losses in large concurrent situation (e.g., 40 to 1 at here). Therefore, we just focus on the problem-prone points of these two extreme cases, and show the test results in Fig. 6. Fortunately, in both two extreme cases, packet loss never happens, and the link utilization always remains at a high level as well. Besides, in other cases of the number of senders (are not shown in Fig. 6), the effect is even more obvious. In TCP-TRIM, since any measured RTT greater than K promptly triggers a backoff action, it is more sensitive to network congestion and can control the queue buildup quickly,

which in turn contributes to the satisfied performance when transferring the successive traffic.

Nonetheless, sometimes we can not determine K from Eq. (26) when Eq. (27) can not be satisfied, which means the switch buffer size is too small to meet Eq. (28) or Eq. (29). For example, with 1 Gbps links and 200 μs RTT, the switch buffer size should be no less than 52 KB to satisfy Eq. (27), while for 10 Gbps links and 100 μs RTT, the swith should provide 331 KB buffer at least. In fact, the popular productive data center switch, like Cisco CAT4948, Triumph, etc., often offers more buffer space for each port [3]. Hence Eq. (27) can be established in most instances. Even if it can not be satisfied, we also suggest that K should be determined based on Eq. (25) since packet loss is the main reason of TCP timeout, which is just the culprit of throughput collapse.

B. Component for Supporting High Concurrency

For providing HTTP-based service, the responses calculated by the intra-rack workers are often sent back to the front-end server in the same rack at nearly the same time, which brings huge pressure to the switch buffer, especially when plenty of workers (more than twenty in a rack) involve in this process. Besides, along with incorrectly inheriting the window size of previous PT, the transfer performance of concurrent HTTP connections gets substantially worse as the "many-to-one" scenario scales up.

To solve this problem, we design a asynchronous timer to desynchronize the concurrent PT transfer. Specifically, when a new PT arrives, both its detection packets and the remained ones are sent out after waiting for a random time duration. This action lowers the arrival intensity of concurrent packets to some extent, and in turn leaves more time to transmit the packets queueing at the switch output port.

In the TCP-TRIM control loop, there are two time points that should start the asynchronous timer. The first one is the time for transferring the probe packets, called "asynchronous detection (AD)", while the other one is "asynchronous stream (AS)", which is the time on receiving the ACKs of probe packets.

***AD Timer*:** Before a connection starts to send the probe packets of a new PT, the TCP-TRIM sender actively waits for a time (t_{AD}) to desynchronize the detection process. In our method, t_{AD} is randomly selected from the interval [0, RTT_{max}], wherein RTT_{max} is the maximum round trip time of HTTP connection so far, and obtained by sensing RTT at sender. Since RTT_{max} indicates the most heavy congestion experienced by the connection so far, each connection starts probing after waiting for RTT_{max} at most, thus as far as possible avoids worsening the network congestion.

***AS Timer*:** After the ACKs of probe packets are received, again the sender waits for a time duration t_{AS} before the following data transfer. t_{AS} is also randomly selected from an interval [0, t_{AS}_max], wherein t_{AS}_max is the maximum waiting time, and obtained based on the previous probe packets. Next, we discuss how to determinate t_{AS}_max.

Suppose that N persistent TCP connections are maintained between N web servers and a single front-end server. Each connection is going to send packets with the congestion window size of W via a bottleneck link whose capacity is C (measured in packets/second). The buffer size at the output port is B (measured in packets), and the link latency is D (measured in seconds). Since the start time of each connection belongs to [0, t_{AS}_max], the duration that NW packets arrive at the switch is t_{AS}_max+W/C. Then we get the arrival rate C_λ of the total NW packets by

$$C_\lambda = \frac{NW}{t_{AS}\_max + \frac{W}{C}}. \tag{30}$$

Additionally, we also get FP_{AS}, the number of flying packets at $t_{AS}\_max + W/C$, by

$$FP_{AS} = (C_\lambda - C)\left(t_{AS}\_max + \frac{W}{C}\right). \tag{31}$$

If no packets are dropped when the NW packets arrive, we have $FP_{AS} \leq B + C(t_{AS}\_max + W/C)$, and then with Eqs. (30) and (31) we get

$$t_{AS}\_max \geqslant \frac{(N-2)\,W - B}{2C}. \tag{32}$$

On the other hand, transfer without queueing means that C_λ should be less than or equal to C. However, $C_\lambda < C$ indicates that the bottleneck link is underutilized and should be avoided, then with Eq. (30) we get

$$t_{AS}\_max \leqslant \frac{(N-1)\,W}{C}. \tag{33}$$

Therefore, to guarantee no packet losses and full bottleneck link utilization as well, t_{AS}_max should simultaneously satisfy Eqs. (32) and (33).

Finally, due to the operation of random selection, the asynchronous timer might still generate some concurrency even if assigning a larger selective interval. Therefore, in our scheme we conservatively stipulate that each connection does not start the remained data transfer until waiting for a time duration t_{AS}, which is randomly selected from the interval $[0, (N-1)W/C]$.

C. Implementation

We implement the design of TCP-TRIM which controls the congestion window at endpoint through RTT measurement. There are three key steps in our design. The first one is RTT measurement, which requires server to provide high-resolution timer (i.e., up to microsecond level) at high data rate and low-latency data center network. Fortunately, the option of high-resolution timer has been provided in the version of 2.6 Linux kernel and later. We simply use this timer to obtain accurate RTT.

The second one is that the calculated window size will be very small or even negative if the measured RTT is very large, i.e., larger than $2 \times min\_RTT$. In TCP-TRIM, we set the minimum value of congestion window to 2, as the same value of that in legacy TCP protocols.

The third issue is the newly arrived PT may be very small, i.e., one or two packets. In our implementation, if the outgoing PT has only 1 packet or 2 packets, the TCP-TRIM sender will still send them to detect congestion and make the congestion window regulation based on Eq. (1).

In addition, to estimate N in Eq. (33), we give a simple method as following. In a sense, $probe\_RTT \times C$ indicates the number of flying packets before the probe packets arrive at the switch output port. For these flying packets we could heuristically assume that they are all probe packets coming from different connections. Hence N could be considered as $probe\_RTT \times C/2$ because each connection sends only two probe packets for each new PT. Note that we do not claim that this method can reach an accurate and correct N, actually the calculated N is often larger than the real value. A larger N does not significantly affect the desynchronization since Eq. (33) is established for getting an interval to randomly assign a start time for transferring the remained packets, instead of determining the precise start time.

IV. Performance Evaluation

In this section, we first run simulation tests to explain how TCP-TRIM avoids the performance impairments described in Section II. B. Then we examine the basic properties of the TCP-TRIM algorithm, such as switch queue length, throughput, convergence, and fairness. Next, we make performance comparison between TCP-TRIM and two data center transport protocols, DCTCP and L^{2}DCT [29] in a fat-tree network [30]. Finally, the implementations on the real testbed are given to evaluate the TCP-TRIM performance in the real web service scenario. Unless otherwise noted, K is set according to Eq. (26), and α, the weight for the new RTT sample during the smoothing process, is set to 0.25 throughout all the tests.

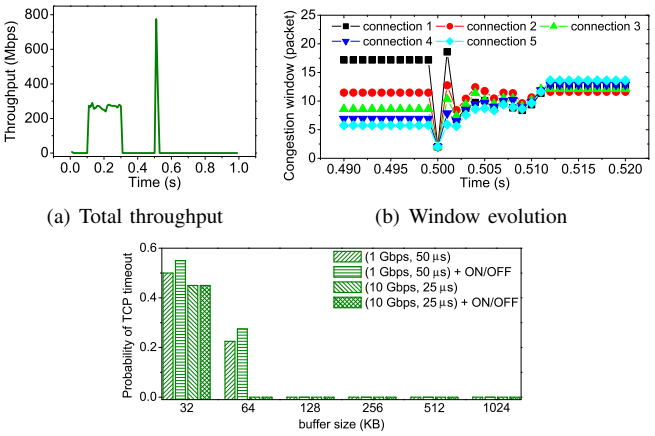
A. Impairments Test

In this part, we use TCP-TRIM to run the simulation tests described in Section II. B, and examine if the problem can be solved effectively.

In Fig. 7(a), we observe that only one spike appears, and the total throughput rapidly reaches to about 800 Mbps at 0.5 s. TCP timeout does not happen, and all the data transfers finish before 0.6 s. The trace also shows the queue length never exceeds 20 packets, which is much less than the buffer size (100 packets), thus no packet is dropped. Additionally, Fig. 7(b) shows that TCP-TRIM strictly limits the window increase during the 200 response transfers so that the window size of each connection never exceeds 20 packets before 0.5 s. When the LPT arrives, the data packets for probing the network congestion are sent out, and each window size suddenly plummets to a very low value (2 packets). After the ACKs of probe packets are received, the sender estimates the current congestion level, and tunes the inherited window size to an appropriate value. Besides, in Fig. 7(c), the probability of TCP timeout with TCP-TRIM is much lower than those of TCP, DCTCP, and L^{2}DCT across all the cases (the results of other protocols are shown in Fig. 5). TCP timeout does not happen in most cases, and the data transfer is basically not influenced by the ON/OFF traffic pattern.

B. TCP-TRIM Properties

For evaluating the particular aspects of TCP-TRIM performance, we set up a simulation scenario as follows. Multiple sender hosts connect to a single receiver server via a switch with 100 packet buffer. All the links are 1 Gbps with 50 μs latency. Each sender host establishes one persistent connection to the receiver.

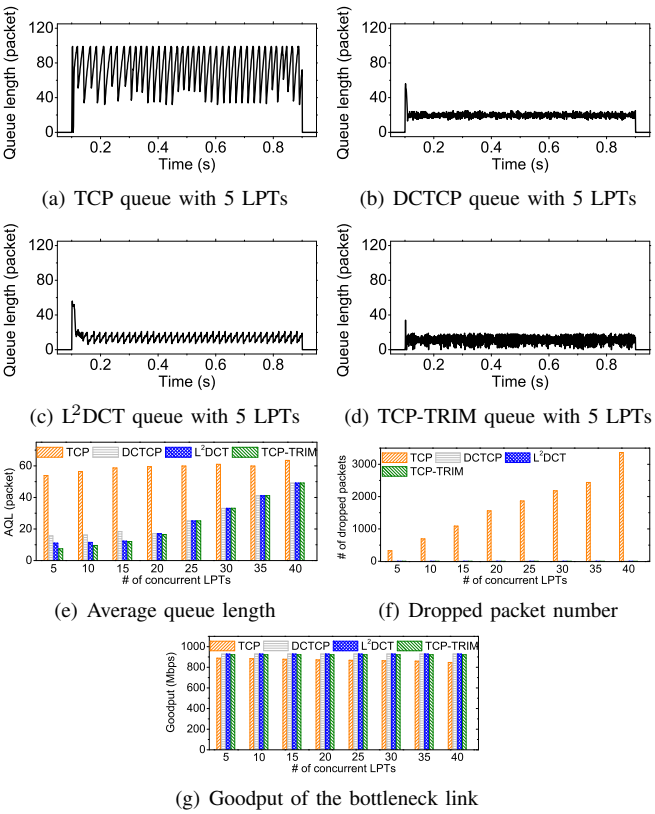


(a) Total throughput (b) Window evolution

(c) Control the "ON/OFF"

Fig. 7. TCP-TRIM's impairment test.

Queue length: to find out whether TCP-TRIM can effectively control the switch queue length, five hosts send 5 LPTs to the receiver from 0.1 s to 0.9 s.



(a) TCP queue with 5 LPTs (b) DCTCP queue with 5 LPTs

(c) L^{2}DCT queue with 5 LPTs (d) TCP-TRIM queue with 5 LPTs

(e) Average queue length (f) Dropped packet number

(g) Goodput of the bottleneck link

Fig. 8. TCP-TRIM properties.

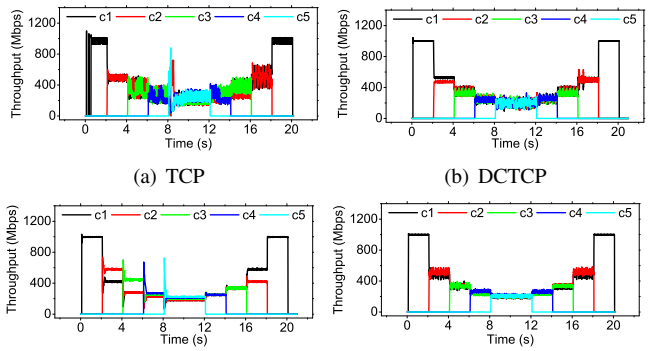
From Fig. 8(a), we observe that the saw-tooth behavior of queue length is obvious. The TCP queue frequently touches the upper boundary of switch buffer, which implies some packets are dropped and TCP timeouts may come together as well. In Fig. 8(b), the queue of DCTCP is well controlled, and the switch queue length is kept about 20 packets, which is

just the preset value of mark threshold in DCTCP. The queue built in L^{2}DCT also shows the similar situation compared with DCTCP. Since they all use 20 packets as the mark threshold used by switch, their switch queue lengths are also almost the same, just as shown in Fig. 8(c). In contrast, except TCP, the queue lengths of other protocols are more stable and smaller, which contributes to the lower packet loss ratio.

Fig. 8(e) shows the average queue length (AQL) under different number of concurrent LPTs. To avoid the impact of TCP timeout, we set RTO at 1 ms to reduce the pause time. From the results, we observe that AQLs of all the protocols show the gradually rising trend as the number of concurrent LPTs increases.

However, AQL of TCP is much higher than those of other protocols throughout all the cases. Overall, TCP-TRIM performs better. We also record the number of dropped packets. In Fig. 8(f), the dropped packet number of TCP becomes larger as the number of concurrent LPTs increases, while for other protocols, no drops and no TCP timeouts happen.

Goodput of the bottleneck link: from Fig. 8(g), we observe that the goodput of TCP is the lowest across all the cases, while other protocols have the similar performance. Their bottleneck link utilizations is nearly 98% as well. Meanwhile, the nearly full bottleneck link utilization in turn testifies the analysis of K configuration described in Section III.



(a) TCP (b) DCTCP

(c) L^{2}DCT (d) TCP-TRIM

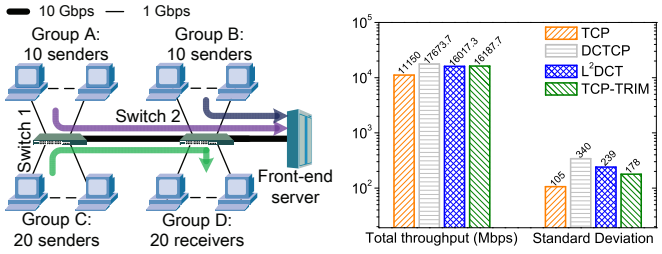
Fig. 9. Convergence test ("c1, c2..." means "connection 1, connection 2...").

Fairness and convergence: in order to test if TCP-TRIM can quickly converge to the fair share, six servers are linked to a switch with a buffer of 100 packets. The link between the receiver (a selected server acts as the front-end) and the switch is with 1 Gbps capacity and 50 μs latency, while the remained links are with 1.1 Gbps and the same latency. In addition, 5 TCP connections are set up before the data transmission happens, and they are kept throughout the whole test. From 0.1 s, we start to send a LPT and then sequentially begin to send other 4 LPTs with 2 s time interval. From 12.1 s, we stop these LPTs one by one using the same interval. The throughputs of connections are depicted in Fig. 9.

From the results, we can observe that TCP-TRIM benefits a lot from its better queue control. Altering the intensity of input traffic does not greatly disorganize the bandwidth share of TCP-TRIM. Consequently, each of the five connections converges to their fair share quickly. For TCP and DCTCP, although their throughputs are approximately fair on average, the convergence process shows large variation throughout the whole test. Since L^{2}DCT is not a completely fair protocol, it performs the worst.

Multi-hop networks: to evaluate TCP-TRIM's performance in a multi-hop, multi-bottleneck environment, we build up a simulated network whose topology has been used in [3]. This topology could also be considered as a part of the leaf-spine topology, which is very prevalent in the commodity data center [39], [40], [41], [42]. Although the cloud service providers often tend to use high speed devices and redundant network architectures to further expand the network capacity, especially for the rack-to-rack communication. However, in some DCs, a significant fraction of core links appear to be persistently congested, but there is enough spare capacity in the core to alleviate congestion [1] [43].

As shown in Fig. 10(a), both Groups A and B have 10 senders, and they all send LPTs to a front-end server. Meanwhile, each sender in Group C also sends a LPT to a receiver selected from Group D. There are two bottlenecks in this topology: both the 10 Gbps link between switch 1 and switch 2 and the 10 Gbps link between switch 2 and the front-end server are oversubscribed. Except the 2 bottleneck links, the remained links are with 1 Gbps bandwidth. The PTs from group A encounter all the bottlenecks.



(a) Multi-hop topology (b) Performance Comparison

Fig. 10. Multi-hop scenario.

TABLE I
THE AVERAGE THROUGHPUT OF EACH GROUP.

Protocol	Group A	Group B	Group C
TCP	259.6 Mbps	471.2 Mbps	233.9 Mbps
DCTCP	139.27 Mbps	814.24 Mbps	406.93 Mbps
L^{2}DCT	237.93 Mbps	692.52 Mbps	335.64 Mbps
TCP-TRIM	342.7 Mbps	638.39 Mbps	318.84 Mbps

We measure the throughput of each flow and calculate the average throughput of each group. The results are shown in Table I. We also calculate the total throughput and the standard deviation (SD) of the average throughput of each group for each protocol. The standard deviation describes the gap of the average throughput between groups. As shown in Fig. 10(b), since frequent buffer overflows cause plenty of drops and TCP timeouts, TCP gets the lowest total throughput. DCTCP has the highest total throughput and standard deviation. Overall, TCP-TRIM performs slightly better than others.

When dealing with the successive traffic, the fundamental difference between TCP, DCTCP, and TCP-TRIM is the

manner of shrinking the congestion window in the congestion avoidance phase. Specifically, in each RTT, TCP always halves the congestion window size once perceiving any packet loss, while DCTCP at most cuts window in half, resulting in the stronger competitiveness compared to TCP. TCP-TRIM also cuts window in half at most. However, as a delay-based scheme, its congestion-sensitivity makes it become more gentle when grabbing the available bandwidth compared with DCTCP. In addition, as L^{2}DCT is a variant of DCTCP, it weakens the competitiveness of long flow, hence it's also less competitive than DCTCP in this scenario. For these reasons, the total throughput of DCTCP is the highest.

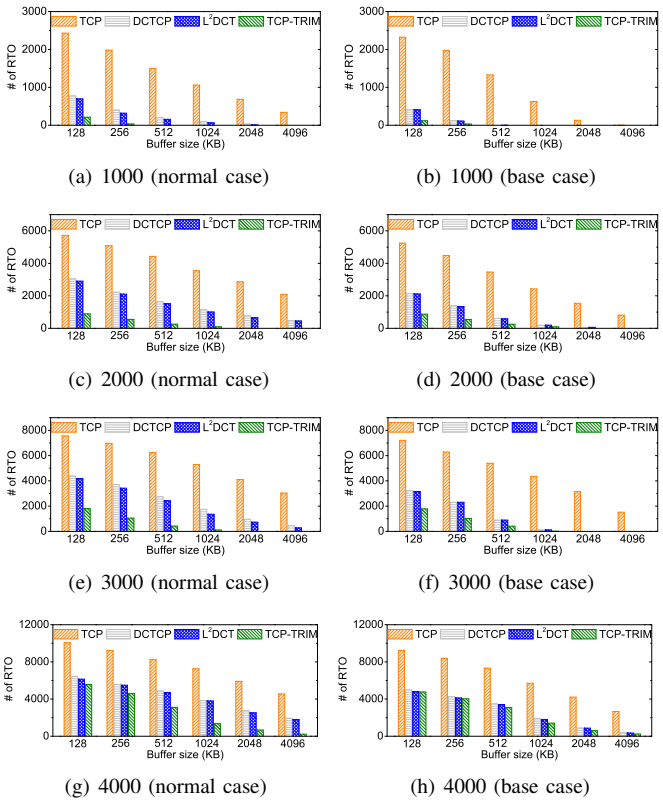
On the other hand, in this scenario, group A first shares bandwidth with group C, then competes with group B, while both group B and group C only share bandwidth with group A, hence group A is the weakest. In view that DCTCP has the strongest competitiveness compared with other protocols, group A is robbed more bandwidth by other two groups when running DCTCP, which results in the highest standard deviation of DCTCP.

C. High Concurrency Test

In this section, we run tests in a high concurrency scenario in which 40 sender servers altogether send traffic to a single receiver server via a switch. The links between the senders and the switch are with 1 Gbps and 50 μs latency, while the link between the switch and the single receiver is with 10 Gbps and 25 μs latency. Each sender server establishes multiple persistent connections, and each connection totally transfers 512 KB data that has been divided into 64 responses. The semisynthetic traffic of each connection is derived from the productive workload described in Section II-A. The arrival order of responses in each test is also randomly generated. Besides, in view that the number of concurrent flows would be less than 8000 even for a very heavily loaded leaf switch in a typical commodity DC [39], we set the number of connections hosted by each sender in this scenario as 25, 50, 75, and 100, respectively. Therefore, the total number of concurrent connections is 1000, 2000, 3000, and 4000 accordingly. We also run tests by varying the switch buffer size, such as 128, 256, 512, 1024, 2048, and 4096 KB.

To evaluate the performance, we trace the total number of Retransmission Time Out (RTO). Since each connection discretely transfers 64 responses, and the window size of the previous response will be retained when the current response starts, this can be treated as the "normal case" in the scenario of typical HTTP traffic. However, to make a comprehensive comparison, we also run tests in the scenario of "base case", which means the connection would terminate and the TCP-SYN handshake would be needed to resume communication when a response finishes.

In Fig. 11(a) and Fig. 11(b), TCP has the highest number of RTO both in the "normal case" and in the "base case". Even with 4 MB buffer size, TCP still cannot eliminate the RTO completely in the "normal case". DCTCP and L^{2}DCT have the similar performance, and they all perform better in the "base case". TCP-TRIM always performs the best. The RTO never occurs when the buffer size is larger than 512 KB in both



(a) 1000 (normal case) (b) 1000 (base case)

(c) 2000 (normal case) (d) 2000 (base case)

(e) 3000 (normal case) (f) 3000 (base case)

(g) 4000 (normal case) (h) 4000 (base case)

Fig. 11. The number of RTO in different number of concurrent connections.

cases. As the number of concurrent connections increases, the performance of each protocol gets worse. TCP-TRIM needs at least 2 MB buffer size to avoid RTO in the cases of 2000 and 3000 connections, as shown in Figs. 11(c)-11(f). When the number of connections is further increased, such as the case of 4000 connections in Fig. 11(g) and Fig. 11(h), TCP-TRIM can not avoid RTO completely either.

Compared with the "base case", the "normal case" means that each connection is long-lived, and the congestion window in the previous ON period is directly inherited when the connection enters the ON period again. From the results of "base case" in Fig. 11, we observe that the number of RTOs in the "normal case" is lowered to some extent across all protocols. The reason is, in the "base case" scenario, establishing connection for each response makes all the protocols enter the ON period with using the initial window (e.g., 2 packets), which is the smallest congestion window in the TCP control loop. Besides, since each protocol in our test has the same initial window, this operation also narrows the performance gap between protocols. Nonetheless, the asynchronous component in our design still helps TCP-TRIM perform a little better than others, although the performance gap is further shrunk with increasing the number of concurrent connections.

D. Performance in the "fat-tree" network

To further understand the performance of TCP-TRIM in typical data center scenario, we set up the popular "fat-tree" topology network [30] to make a comprehensive performance comparison. The parameter settings in DCTCP and L^{2}DCT are in line with [3] and [29] respectively.

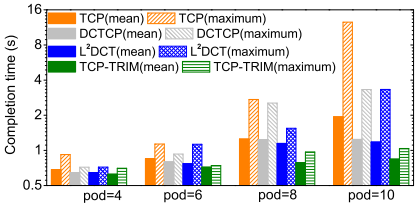


Fig. 12. Completion times in 10 Gbps fat-tree network.

TABLE II
THE NUMBER OF TIMEOUTS IN EACH PROTOCOL.

Pod number	TCP	DCTCP	L^{2}DCT	TCP-TRIM
4	13	9	9	8
6	85	75	71	39
8	452	440	274	141
10	1738	859	493	285

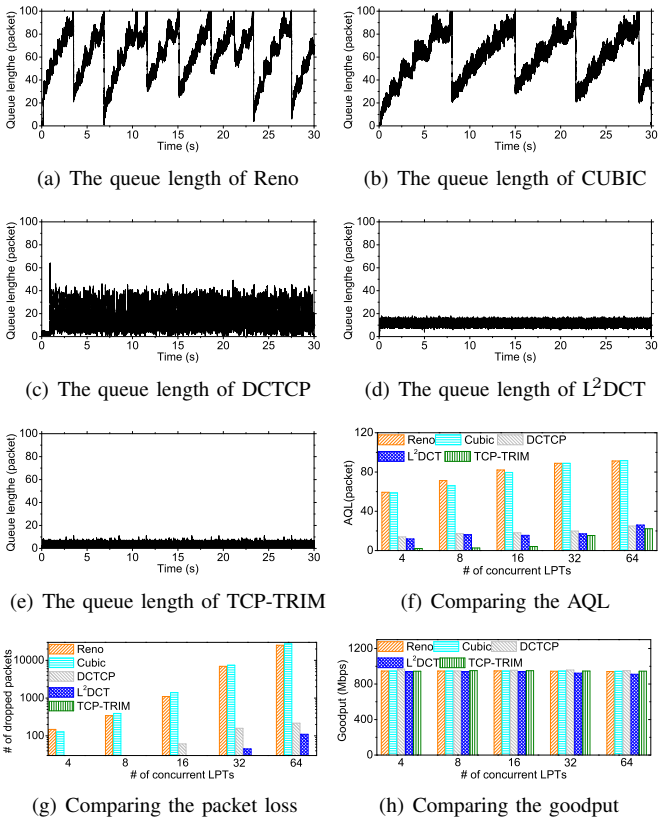
In this scenario, each server totally sends 1 MB data on a persistent HTTP connection to a randomly selected sink server which acts as the front-end. The 1 MB data are artificially divided into some small objectives (from 2 KB to 6 KB) and a big one (the remained data) in advance. Small objectives start at 0.1 s, while the big one is sent from 0.5 s. We calculate the mean of completion times of all the servers and also give the maximum sample value in different network scale (pod number is from 4 to 10). The link bandwidth and switch buffer size are set as 10 Gbps and 350 KB respectively.

Fig. 12 shows that TCP always gets the worst performance in all cases. As the network scales up and more workload involves in, the tail completion times of TCP rise sharply. On the other hand, other schemes perform better, either in getting small mean completion time or in cutting the tail. DCTCP employs Explicit Congestion Notification (ECN) to control the switch queue length thus avoiding packet loss and TCP timeout. L^{2}DCT still uses ECN, and also weights flow to further smooth the increase in congestion window. However, just like TCP, both of the two protocols are unaware of the problem of window inheritance on persistent HTTP connection thus failing to actively limit the expansions of their windows. As a whole, ascribing to the moderate window inheritance and the timely delay-based queue control, TCP-TRIM performs the best across all the test cases, and the advantage is more significant as the number of pod increases. For further testifying our observations, we also record the total number of timeouts of each protocol in each test case. In Table II, TCP still has the most timeouts, and is followed by DCTCP and L^{2}DCT. TCP-TRIM always gets the least timeouts, especially when pod number is 10, the improved ratio compared to TCP is up to 80%.

E. Real Implementation

In this section, we use several DELL OptiPlex 3010 Desktop machines, which act as the back-end servers, to observe the performance of TCP-TRIM in the real testbed. These machines connect to the front-end server (CPU: Intel XEON E5-2650, MEMORY: 24G) via a switch with 100 Mbps and 1 Gbps links, respectively. The kernel patches for supporting DCTCP, L^{2}DCT, and TCP-TRIM are pre-installed into all the servers.

Firstly, we build a 2-to-1 scenario (two back-end servers send data to the front-end server), and use Iperf [38] to create multiple processes for generating long-lived traffic. specifically, each machine sends 2, 4, 8, 16, and 32 LPTs, respectively, hence the number of concurrent LPTs is 4, 8, 16, 32, and 64 accordingly. We show the switch queue length when the LPT number is 4, and calculate the Average Queue Length (AQL) of each test case. We also show the number of dropped packets and overall goodput of each protocol.



(a) The queue length of Reno (b) The queue length of CUBIC

(c) The queue length of DCTCP (d) The queue length of L^{2}DCT

(e) The queue length of TCP-TRIM (f) Comparing the AQL

(g) Comparing the packet loss (h) Comparing the goodput

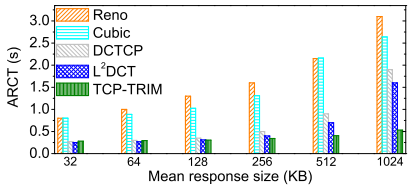
Fig. 13. The queue length, packet losses, and overall goodputs on testbed.

In Fig. 13(a) and Fig. 13(b), the queue lengths of Reno and CUBIC frequently reach the upper bound on the buffer size, which indicates that many packets are dropped. On the contrary, as shown in Figs. 13(c)-13(e), the queues of DCTCP, L^{2}DCT, and TCP-TRIM are well controlled, hence packet loss does not happen. Moreover, the queue length of TCP-TRIM is the smallest, which is further testified in Fig. 13(f). Besides, when running TCP-TRIM, no packets are dropped throughout the whole test, and its goodput is satisfied as well, as shown in Fig. 13(g) and Fig. 13(h).

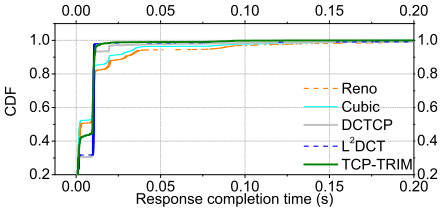
Secondly, we use 100 Mbps links, and let two DELL machines firstly send 2 large files to the front-end persistently. After that, the third one sends 100 responses to the front-end. The data size of each response is randomly generated from the same mean size with 10% variation. In addition, we change the mean response size of each test case from 32 KB to 1 MB. In each test case, we record the completion time of each response and calculate the average response completion time

(ARCT). From Fig. 14, we observe that the ARCTs in all the protocols become larger as the mean response size increases. By contrast, the increasing trend of ARCTs in TCP-TRIM is more gentle. Since TCP-TRIM is a delay-based scheme, its high congestion-sensitivity makes the sender start backoff earlier, hence the switch queue length is maintained at a lower level, which in turn leads to a smaller queueing delay when the third server sends responses. Also, effectively tuning the inherited window size brings further improvement, thus TCP-TRIM performs better than other schemes.

Next, we set up a simple web service scenario, in which 4 DELL machines establish 4 persistent connections and send altogether 4000 responses to the front-end server via five 1 Gbps links. The distributions of response size and time interval are totally in accordance with the description in Fig. 2. We respectively run the test with using Reno, CUBIC, DCTCP, L^{2}DCT, and TCP-TRIM, and record the completion time of each response. In Fig. 14(b), we give the distribution of the completion times of all the responses for each protocol. In this scenario, the discrete response transfer forms the ON/OFF traffic pattern, hence all the protocols must deal with the impact from the inherited window size. However, except from TCP-TRIM, others can not perceive this impact, and they directly use the inherited window size to transfer the newly arriving responses, resulting in potential heavy congestion, which in turn impairs their performance to some extent. Finally, since nearly 99% of the response completion times is below 25 ms, TCP-TRIM performs the best as a whole.



(a) The average response completion time



(b) The CDF of response completion time

Fig. 14. Real Implementation.

V. Related work

In the literature of data center transport mechanism, many recent schemes aim to handle the highly concurrent communication. Among them, the protocols based on explicit feedback, such as [3], [31], and [29], are popular due to their accurate congestion notification [32], [33], [34], [35].

Data Center TCP (DCTCP) [3] leverages Explicit Congestion Notification (ECN) to keep the switch queue length around a given threshold thus alleviating the packet losses and TCP timeouts. D^{2}TCP [31] is proposed based on DCTCP, while considering both the congestion control and deadline requirements by elegantly adjusting the extent of window decreasing. L^{2}DCT [29] still follows the properties of DCTCP in concurrency control while introducing the Least Attained Service (LAS) scheduling at the sender. Unfortunately, we observe that L^{2}DCT fails to distinguish short and long flows because any flow is long-lived with using the persistent TCP connections.

As for the up-to-date delay-based schemes in data center network, both [36] and [37] have experimentally demonstrated that measuring simple packet delay at host is an effective way to obtain the real-time congestion level. Hence they all proposed their own congestion control method that exploits latency-based congestion feedback to keep the delay low while delivering high throughput.

These schemes are effective, and to some extent, can alleviate the network congestion in the scenario of concurrent data transfer. Nonetheless, when dealing with the ON/OFF HTTP traffic, their major issue is that they directly use the expired congestion window (probably does not match the current network state) to send new traffic when the connection state changes from idle to active. Since data center traffic is burst-prone and the network state varies frequently [3], the sender should measure the network state in real time, especially when the connection just leaves the OFF period and enters the ON period. Therefore, perceiving the impacts from transferring ON/OFF traffic on the persistent connections can further improve their performances, which significantly motivates the study reported in this work.

In addition, the studies on the bursty behavior of TCP over HTTP connection have already been conducted in WAN context, such as [18] and [19]. However, in view that the physical environment of DCN is very different from that of WAN, directly using the existing schemes of WAN to resolve the problem of DCN may not work. For example, as described in [18], the existing idle detection schemes in WAN can not perceive an idle period shorter than a RTO, while in DCN plenty of idle periods are far from reaching the level of RTO. Besides, these works do not consider how to control the large-scale concurrent transfer, which is just the unique characteristic of data center network. Moreover, due to a lack of accurate bandwidth detection, the proposed methods for controlling the packet bursts fail to guarantee both high bandwidth and low delay transmission, which is exactly the crucial requirement for designing the data center transport protocol [20], [21]. For instance, both [18] and [24] have suggested that the congestion window should be reset before entering a new ON period. However, we argue that the bottleneck link would be underutilized if the network actually had enough capacity to enable the transfer to start with a large window size.

Overall, to the best of our knowledge, none of the above works specially focuses on handling the improper congestion window evolution on concurrent HTTP connections in intra-data center, which is exactly the goal of our work.

VI. Conclusion

We design and implement TCP-TRIM, a transmission control protocol for HTTP application scenario. By using probe

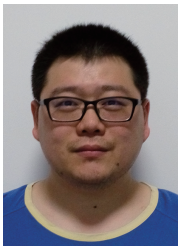
packets and delay-based congestion control, TCP-TRIM greatly improves the transmission performance of concurrent HTTP traffic. Besides, TCP-TRIM is able to well control the switch queue length thus avoiding packet loss and TCP timeout without any hardware refresh. By using at-scale simulations and testbed implementations, we show that TCP-TRIM has better performance (up to 80% reduction in ARCT) than TCP. Future work is the performance evaluation in a large-scale testbed.

ACKNOWLEDGMENT

This work is supported by the National Natural Science Foundation of China (61572530, 61502539, 61402541, 61462007 and 61420106009).

REFERENCES

[1] T. Benson, A. Akella, and D. Maltz, Network Traffic Characteristics of Data Centers in the Wild, *in Proc. IMC*, 2010, pp. 267-280.
[2] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken, The Nature of Datacenter Traffic: Measurements & Analysis, *in Proc. IMC*, 2009, pp. 202-208.
[3] M. Alizadeh, A. Greenberg, D. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, Data Center TCP (DCTCP), *in Proc. ACM SIGCOMM*, 2010, pp. 63-74.
[4] s. Bensley, L. Eggert, D. Thaler, and G. Judd, Datacenter tcp (dctcp): Tcp congestion control for datacenters, *IETF Internet-Draft*, draft-ietf-tcpm-dctcp-04, 2017.
[5] A. Greenberg, J.R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta, VL2: A Scalable and Flexible Data Center Network, *in Proc. ACM SIGCOMM*, 2009, pp. 51-62.
[6] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica, Managing Data Transfers in Computer Clusters with Orchestra, *in Proc. ACM SIGCOMM*, 2011, pp. 98-109.
[7] http://pages.cs.wisc.edu/tbenson/IMC10_Data.html, 2017.
[8] http://hadoop.apache.org/docs/current/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html, Jan. 20, 2017.
[9] http://www.bigsynapse.com/mapreduce-internals, 2017.
[10] T. Garry, T. Deshpande, and S. Karanth, Hadoop: Data Processing and Modelling, *Packt Publishing Ltd*, 2016, charpter 11, pp. 901.
[11] M. Yu, A. Greenberg, J. Rexford, and L. Yuan, Profiling Network Performance for Multi-tier Data Center Applications, *in Proc. NSDI*, 2011, pp. 57-70.
[12] Y. Wang, X. Que, and W. Yu, Hadoop acceleration through network levitated merge, *in Proc. ACM HPCNSA*, 2011, pp. 57-66.
[13] C. Joachim, "HTTP/TCP connection and flow characteristics," *Performance Evaluation*, vol. 42, no. 2, pp. 149-162, 2000.
[14] D. Ersoz, M. S. Yousif, and C. R. Das, Characterizing network traffic in a cluster-based, multi-tier data center, *in Proc. IEEE ICDCS*, 2007, pp. 59-68.
[15] Y. Chen, R. Mahajan, B. Sridharan, and Z. Zhang, A Provider-side View of Web Search Response Time, *in Proc. ACM SIGCOMM*, 2013, pp. 243-254.
[16] J. J. Lee and M. Gupta, A new traffic model for current user web browsing behavior, *in Proc. Intel corporation*, 2007.
[17] H. Choi and J. O. Limb, A behavioral model of web traffic, *in Proc. IEEE ICNP*, 1999, pp. 327-334.
[18] https://tools.ietf.org/html/draft-hughes-restart-00, Dec. 1, 2001.
[19] M. Handley, J. Padhye, and S. Floyd, RFC 2861: TCP Congestion Window Validation, June 2000.
[20] J. Zhang, F. Ren, and C. Lin, "Survey on transport control in data center networks," *IEEE Network*, vol. 27, no. 4, pp. 22-26, Jul. 2013.
[21] S. Prasanthi, and J. Jung, "Transport protocols for data center networks: a survey of issues, solutions and challenges," *Photonic Network Communications*, vol. 31, no. 1, pp. 112-128, 2016.
[22] The Network Simulator–ns-2, http://www.isi.edu/nsnam/ns, 2014.
[23] R. Jain and S. Routhier, "Packet Trains-Measurements and a New Model for Computer Network Traffic," *IEEE Journal of Selected Areas in Communications*, vol. SAC-4, no. 6, pp. 986-995, Sept. 1986.
[24] J. Zhang, F. Ren, L. Tang and C. Lin, Taming TCP Incast Throughput Collapse in Data Center Networks, *in Proc. IEEE ICNP*, 2013, pp. 1-10.
[25] V. Vasudevan, A. Phanishayee, H. Shah, E. Krevat, D. Andersen, G. Ganger, G. Gibson, and B. Mueller, Safe and effective fine-grained TCP retransmissions for datacenter communication, *in Proc. ACM SIGCOMM*, 2009, pp. 303-314.
[26] P. Cheng, F. Ren, R. Shu, and C. Lin, Catch the whole lot in an action: Rapid precise packet loss notification in data centers, *in Proc. USENIX NSDI*, 2014, pp. 17-28.
[27] J. Huang, Y. Huang, J. Wang, and T. He, Packet Slicing for Highly Concurrent TCPs in Data Center Networks with COTS Switches, *in Proc. IEEE ICNP*, 2015, pp. 22-31.
[28] J. Huang, T. He, Y. Huang, and J. Wang, ARS: Cross-layer adaptive request scheduling to mitigate TCP incast in data center networks, *in Proc. IEEE INFOCOM*, 2016, pp. 1-9.
[29] A. Munir, I. A. Uzmi, A. Mushtaq, S. N. Ismail, M. S. Iqbal, B. Khan, Minimizing flow completion time in data centers, *in Proc. IEEE INFOCOM*, 2013, pp. 2157-2165.
[30] Y. Zhang, and N. Ansari, On Architecture Design, Congestion Notification, TCP Incast and Power Consumption in Data Centers, *IEEE Communications Surveys & Tutorials*, vol. 15, no. 1, pp. 39-64, First quarter 2013.
[31] B. Vamanan, J. Hasan, and T. N. Vijaykumar, Deadline-Aware Datacenter TCP (D^2TCP), *in Proc. ACM SIGCOMM*, 2012, pp. 115-126.
[32] J. Wang, P. Dong, J. Chen, J. Huang, S. Zhang, and W. Wang, "Adaptive explicit congestion control based on bandwidth estimation for high bandwidth-delay product networks," *Comput. Commun.*, vol. 36, no. 10, pp. 1235-1244, Jun. 2013.
[33] W. Jiang, F. Ren, and C. Lin, "Phase Plane Analysis of Quantized Congestion Notification for Data Center Ethernet," *IEEE/ACM Trans. Networking*, vol. 23, no. 1, pp. 1-14, Feb. 2015.
[34] T. Zhang, J. Wang, J. Huang, Y. Huang, J. Chen, and Y. Pan, "Adaptive-Acceleration Data Center TCP," *IEEE Trans. Comput.*, vol. 64, no. 6, pp. 1522-1533, Jun. 2015.
[35] T. Zhang, J. Wang, J. Huang, Y. Huang, J. Chen, and Y. Pan, "Adaptive marking threshold method for delay-sensistive TCP in data center network," *Journal of Network and Computer Applications*, vol. 61, pp. 222-234, Feb. 2016.
[36] R. Mittal, V. T. Lam, N. Dukkipati, E. Blem, H. Wassel, M. Ghobadi, A. Vahdat, Y. Wang, D. Wetherall, and D. Zats, TIMELY: RTT-based Congestion Control for the Datacenter, *in Proc. ACM SIGCOMM*, 2015, pp. 537-550.
[37] C Lee, C Park, K Jang, S Moon, and D Han, Accurate Latency-based Congestion Feedback for Datacenters, *in Proc. USENIX ATC*, 2015, pp. 403-415.
[38] A. Tirumala and L. Cottrell, Iperf Quick Mode, http://www-iepm.slac.stanford.edu/bw/iperf res.html.
[39] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, V. T. Lam, F. Matus, R. Pan, N. Yadav, and G. Varghese, CONGA: Distributed congestion-aware load balancing for datacenters, *in Proc. ACM SIGCOMM*, 2014, pp. 503-514.
[40] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker, pFabric: Minimal Near-Optimal Datacenter Transport, *in Proc. ACM SIGCOMM*, 2013, pp. 435-446.
[41] Y. Zhao, K. Chen, W. Bai, M. Yu, C. Tian, Y. Geng, Y. Zhang, D. Li, and S. Wang, RAPIER: Integrating Routing and Scheduling for Coflow-aware Data Center Networks, *in Proc. IEEE INFOCOM*, 2015, pp. 424-432.
[42] W. Bai, L. Chen, K. Chen, D. Han, C. Tian, and H. Wang, Practical Information-Agnostic Flow Scheduling in Data Center Networks, *in Proc. USENIX NSDI*, 2015, pp. 455-468.
[43] D. Zhuo, Q. Zhang, V. Liu, A. Krishnamurthy, and T. Anderson, RackCC: Rack-level Congestion Control, *in Proc. Hotnets*, 2016.



Tao Zhang is pursuing his Ph.D. degree in the Department of Information Science and Engineering, Central South University, China. His research interests include congestion control and data center networks.

JianXin Wang received the BEng and MEng degrees in computer engineering from Central South University, China, in 1992 and 1996, respectively, and the PhD degree in computer science from Central South University, China, in 2001. He is the chair of and a professor in Department of Computer Science, Central South University, Changsha, Hunan, P.R. China. His current research interests include algorithm analysis and optimization, parameraized algorithm, Bioinformatics and computer network. He is a senior member of IEEE.

Jiawei Huang obtained his PhD (2008) and Masters degrees (2004) from the School of Information Science and Engineering at Central South University. He also received his Bachelors (1999) degree from the School of Computer Science at Hunan University. He is now an associate professor in the School of Information Science and Engineering at Central South University, China. His research interests include performance modeling, analysis, and optimization for wireless networks and data center networks.

Jianer Chen received the PhD degree in computer science from Courant Institute of New York University in 1987 and the PhD degree in mathematics from Columbia University in 1990. He is currently a professor of computer science at Texas A&M University, and Central South University in Changsha, Hunan, P.R. China. His main research is centered on computer algorithms and their applications. His current research projects include exact and parameterized algorithms, computer graphics, computer networks, and computational biology.

Yi Pan received his B.Eng. and M.Eng. degrees in computer engineering from Tsinghua University, China, in 1982 and 1984, respectively, and his Ph.D. degree in computer science from the University of Pittsburgh, USA, in 1991. He is currently a Professor and Chair of the Department of Computer Science and Professor of the Department of Computer Information Systems at Georgia State University. His research interests include parallel and cloud computing, wireless networks, and bioinformatics. He is a senior member of the IEEE.

Geyong Min received the B.Sc. degree in computer science from Huazhong University of Science and Technology, Wuhan, China, in 1995 and the Ph.D. degree in computing science from the University of Glasgow, Glasgow, U.K., in 2003. He is a Professor of high-performance computing and networking with the Department of Mathematics and Computer Science, College of Engineering, Mathematics and Physical Sciences, University of Exeter, Exeter, U.K. His research interests include future Internet, computer networks, wireless communications, multimedia systems, high-performance computing, modeling, and performance engineering.