Tuning the Aggressive TCP Behavior for Highly Concurrent HTTP Connections in Data Center

Jiawei Huang, Jianxin Wang, Tao Zhang
School of Information Science and Engineering
Central South University
ChangSha, China
Emails: {jiaweihuang, jxwang, taozhang}@csu.edu.cn

Jianer Chen
Dept. of Computer Science
Texas A&M University
College Station, USA
Email: chen@cse.tamu.edu

Yi Pan
Dept. of Computer Science
Georgia State University
Atlanta, USA
Email: yipan@gsu.edu

***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 first reveal that the TCP's aggressive behavior in increasing congestion window causes TCP timeouts and throughput collapse. 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 show 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 performances directly impact the enterprise revenue, the network operators try their best to shorten the service response time thus providing end-users with good experiences [3]. 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 [4]. 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 [5].

On the other hand, 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. 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 [6]. To achieve fast response 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 [7]. Previous research has reported that, HTTP 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) [8], [9]. However, there are two key factors 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 [9], [10]. 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 [7]. 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]. Such many-to-one traffic patterns, joint with the droptail queue management of switch buffer, brings 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 performances.

Existing data center network TCPs have developed miscellaneous schemes to alleviate congestion of concurrent TCP flows. When transferring HTTP traffic, however, they share a 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 frequent overflows, resulting in TCP timeout and throughput collapse.

In this work, based on the typical ON/OFF HTTP workload, we emphatically 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 response transfers, our design also regulates the congestion window size according to the end-to-end delay. The contributions are as follows:

• We provide the first extensive study to exploit the root cause of performance degradation of highly concurrent HTTP connections. We reveal the impact of HTTP's ON/OFF style on TCP protocol 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 increasing of 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 to reduce congestion window in TCP-TRIM.

• We evaluate the performance of TCP-TRIM by using at-scale NS2 [11] 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 brings remarkable revenue (i.e., up to 80%) in reducing the average completion time of HTTP response.

The remainder of the 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. At last, 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 the root reason why current 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) [8], 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], [7].

For comprehensively understanding the characteristic of HTTP traffic, we recorded the real workload of data center hosted in university campus. From 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]. To describe the ON/OFF pattern more clearly, we define a packet train (PT) as a burst of packets on a 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 [12].

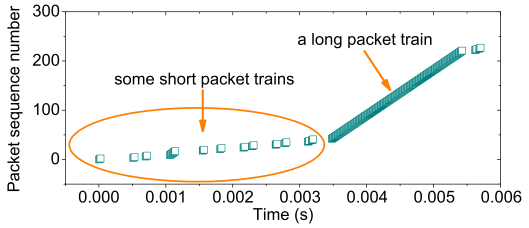


Fig. 1. Understand the "Packet Train".

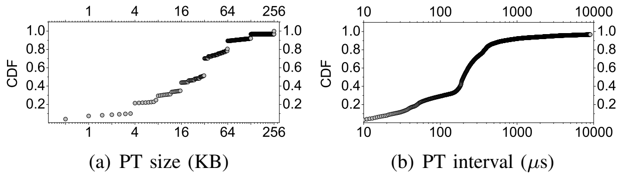


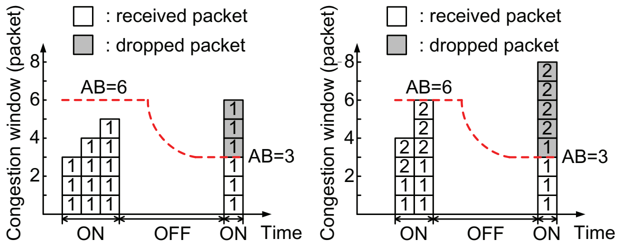
Fig. 2. CDFs of PT size and interval.

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, it contributes much more to the increase of packet sequence number compared to 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. On the other hand, 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. We give the detailed description as following.

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 single flow [9]. 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.

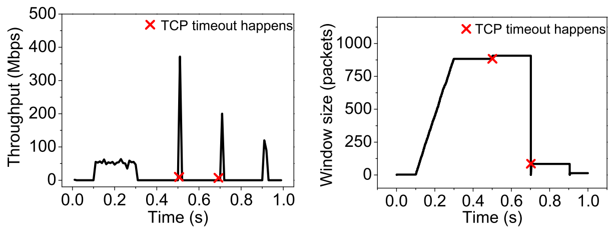


(a) Single connection (b) Multiple connections

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 new connections (are not shown in Fig. 3(a)) 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 connections, this impairment becomes severer. As shown in Fig. 3(b), two coexisting connections cause all the packets in the congestion window of connection 2 are dropped, resulting in TCP timeout [13].

For further elaborating this impairment, we install the synthetic traffic derived from our real trace data on a many-to-one scenario built by NS2. Specifically, five servers connect to a front-end server via a switch with 100 packets buffer through five 1 Gbps links with 50 μs latency. Each server first receives 200 requests, and then returns 200 responses from 0.1 s. Each response has the data size ranging from 2 KB to 10 KB, and the interval between two neighboring responses is randomly generated based on 1 ms mean. 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. Meanwhile, we keep the 5 TCP connections throughout the whole transmission.



(a) Throughput (b) Congestion window

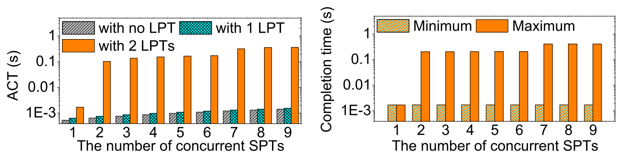
Fig. 4. Throughput and window size of connection 5.

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 connection 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. For making it clear 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, in our trace the inherited window sizes in connection 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.

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 last ON period and sends as much 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.

2) Performance Impairment 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.

For further illustrating this issue, we rebuild the previous many-to-one scenario, and respectively start 0, 1 and 2 LPTs from 0.1 s to the end. The other web servers concurrently burst SPTs (each with 10 packets) at 0.3 s. The switch buffer size is set to 100 packets, and the retransmission timeout (RTO) is 200 milliseconds. We gradually change the number of SPT servers, and use the average completion time (ACT) of PTs as the performance metric.



(a) ACTs of concurrent SPTs (b) Min. & max. completion times

Fig. 5. Concurrency impairment.

Fig. 5(a) shows that the average completion time of SPTs becomes larger with the increasing number of LPTs. Especially in the cases of 2 LPTs, ACTs become unacceptably high. We record the minimum and maximum completion times as shown in Fig. 5(b). It is clear that the maximum completion time of SPT becomes higher as the number of concurrent SPTs increases. Additionally, from the trace, we also find the SPT with the maximum completion time experiences two TCP timeouts when the number of SPTs exceeds 6.

C. Summary

The 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 transfer of HTTP responses. These conclusions motivated 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.

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 three 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, and (iii) 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. In the following, we describe the design detail 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

```
1: Before sending a new packet :
   ( not a retransmission packet )
2: if $ti > smooth\_RTT$ then
3:     Saving the accumulated window size;
4:     $cwnd \leftarrow 2$;
5:     Sending the probe packets in current window;
6:     Suspending the packet transfer;
7: end if
8: 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 redetect congestion and smooth sending rate.

2) Tuning Congestion Window: for each arriving ACK, TCP-TRIM measures the current RTT, and updates three variables by the following 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 following two cases, which is shown in Algorithm 2.

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,

Algorithm 2 ACK action

```
1: For each arriving ACK :
2: $smooth\_RTT \leftarrow (1-\alpha)smooth\_RTT + \alpha RTT$;
3: if $RTT < min\_RTT$ then
4:     $min\_RTT \leftarrow RTT$;
5:     Update $K$;
6: end if
7: if the current ACK belongs to the probe packet then
8:     if it arrives in a smooth_RTT then
9:         $cwnd \leftarrow s\_cwnd(1 - \frac{probe\_RTT - min\_RTT}{min\_RTT})$;
10:        Resume packet transfer;
11:    else
12:        $cwnd \leftarrow 2$;
13:        Resume packet transfer;
14:    end if
15: else
16:    if $RTT \geqslant K$ then
17:        $ep \leftarrow \frac{RTT-K}{RTT}$;
18:        $cwnd \leftarrow cwnd(1 - \frac{ep}{2})$;
19:    end if
20: end if
```

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 [13], just send the new PT with congestion window size 2 to minimizes 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

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

And then we 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 size 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

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

B. Guideline for Choosing K

TCP-TRIM uses threshold K to calculate the sending rate and then controls the queue length on switch buffer. It is a challenge to achieve both the high unitization of bottleneck link and minimum queue length on switch buffer. In this subsection, we introduce how to determine the threshold K by analyzing the steady state behavior of TCP-TRIM.

Suppose that N persistent TCP connections are totally synchronized, and maintained between N web 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

$$Q = C(K - D). \quad (4)$$

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 switch queue length is just equal to Q, and at the same time each PT is in the ith round's transfer, then we get the window size of each PT in the ith round by

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

Since the switch queue length does not exceed Q at the ith round, all PTs increase their congestion window in the $(i+1)$st round to

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

However, the queue length will exceed Q in the $(i+1)$st RTT, with the result that each connection perceives the $(i+1)$st RTT is greater than K and starts to decrease the sending rate. Then the maximum queue length Q_{max} is

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

When the queue length reaches Q_{max} in the $(i+1)$st RTT, we calculate the RTT value of the jth PT as

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

From Equations (2) and (3), we also get the current congestion level by

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

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

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

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

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

By substituting Equations (7) into (11), we get

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

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

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

With Equation (13) and Equation (12), we get

$$\begin{aligned} &C\left(K-D\right)+N > \\ &\left(\frac{CK+N}{2N}\right)\left(N-1+CK\ln\frac{CK+1}{CK+N}\right). \end{aligned} \quad (14)$$

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

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

Meanwhile, we simplify it and get

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

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

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

where 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 has an upper bounder and be a convex function. Then we get

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

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

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

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

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

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

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

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, we get K as

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

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-speed and low-latency data center network. Fortunately, the option of high-resolution timer has been provided in the Linux kernel 2.6 and later version. We simply use this timer to obtain accurate RTT.

The second one is 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 extreme 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 it or them to detect congestion and make the congestion window regulation based on Equation (1).

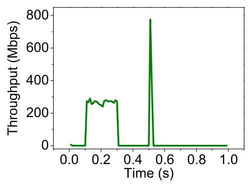
IV. Performance Evaluation

In this section, we first run simulation tests to explain how TCP-TRIM avoids the performance impairments described in Section 2.2. 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. 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 Equation (22), and α, the weight for the new RTT sample during the smoothing process, is set to 0.25 throughout all the tests.

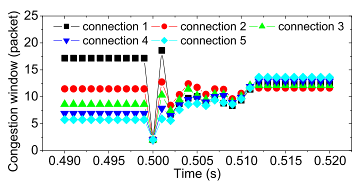
A. Impairments Test

1) Window Smoothing on HTTP Connections: in this section, we examine the modified bandwidth detection behaviour of TCP-TRIM. The detailed scenario has been described in section II.B.

From Fig. 6(a), we observe that there is only one spike and the throughput of bottleneck link rapidly reaches about 800 Mbps at 0.5 s. None of HTTP connections experiences TCP timeouts, and they all finish before 0.6 s. Our trace also



(a) Total throughput



(b) Window evolution

Fig. 6. TCP-TRIM's impairment test.

confirms this because the recorded queue length never exceeds 20 packets, which is much less than the 100 packet switch buffer, thus no packet is dropped. Additionally, the window evolution shown in Fig. 6(b) also helps us understand why TCP-TRIM performs better in controlling the HTTP traffic. Specifically, 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, packet probing starts to work, and each window size suddenly plummets to a very low value (2 packets). After that, the arriving ACKs of probe packets bring the current congestion extent back to the senders, then they tune their inherited window size to an appropriate value and resume their remained data transfers.

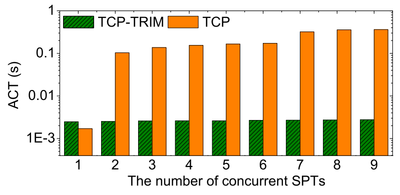
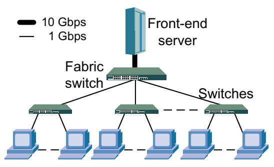


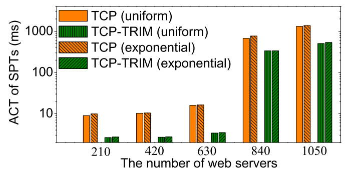
Fig. 7. ACTs of SPTs with 2 LPTs.

2) Handling the Concurrent HTTP Connections: we repeat the test in Fig. 5(a) to test the performance of TCP-TRIM under concurrent HTTP connections. From Fig. 7, we observe that the increase of the number of concurrent PTs does not seriously impact the TCP-TRIM performance. The average completion time (ACT) in each case is only several milliseconds, while TCP's ACT is up to two orders of magnitude except the case of single SPT. The reason is that TCP-TRIM employs delay-based congestion detection to make back-off timely, thus taking up small footprint in the switch buffer and remaining sufficient unused buffer space to absorb the data burst that comes from the high concurrency. Consequently, packet loss and TCP timeout are greatly alleviated, thus helping PTs to obtain less completion times.

Next, to test TCP-TRIM's performance under large-scale HTTP concurrency, we create a simulated network topology as shown in Fig 8(a). Wherein each switch links to 42 servers, and a single front-end server connects with these switches via a fabric switch. All the links have 1 Gbps bandwidth and 20 μs latency, while the cable nearest the front-end has 10 Gbps and 10 μs latency. Within each switch, there are 2 servers maintain two LPTs running throughout the test, and the remained servers transfer the SPTs in the 0.5 s time interval with the uniform and exponential distribution respectively. The destination of all the packets is the front-end server, and the size of PT is determined from the proportion shown in Fig. 2(a). We focus on the overall performance of SPTs since the throughput collapse of LPTs is alleviated by setting a smaller TCP timeout value (20 ms in our tests) as default. During the test, the number of switches at the second level varies from 5 to 25, hence the total number of servers varies from 210 to 1050 accordingly. Each test case is repeated for 100 times to calculate the ACT of SPTs.



(a) Large scale test topology



(b) ACTs of SPTs.

Fig. 8. Large scale scenario.

In Fig 8(b), although both protocols obtain small ACT when the number of web servers is less than 630, TCP-TRIM still reduces the ACT of TCP by up to 80%. Under high concurrency, ACT becomes larger. However, even when the number of web servers exceeds 840, the revenue coming from TCP-TRIM is still about 50%.

B. TCP-TRIM Properties

For evaluating the particular aspects of TCP-TRIM performance, we set up a simulation scenario as following. Multiple servers connect to a front-end server via a switch with 100 packet buffer. All the links are 1 Gbps with 50 μs latency. The servers are the senders, while the front-end server is the receiver. The switch operates in standard drop-tail mode.

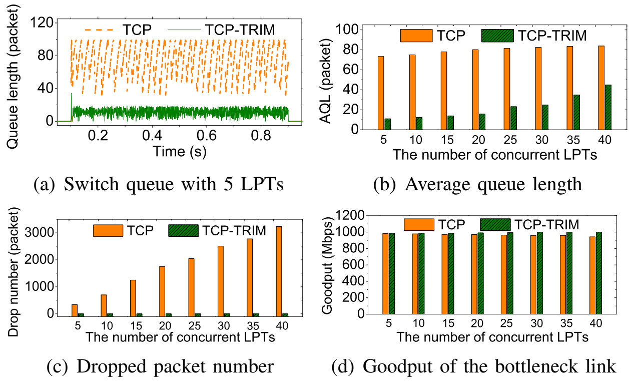
Queue length: to find out whether TCP-TRIM can effectively control the switch queue length, five servers respectively establish 5 persistent connections to the front-end server from 0.1 s to 0.9 s.

From Fig. 9(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 contrast, TCP-TRIM maintains the stable and small queue length.

Fig. 9(b) shows the average queue length (AQL) under different number of concurrent PTs. 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 both TCP and TCP-TRIM show a rising trend as the number of concurrent PTs increases.

However, AQL of TCP is much higher than that of TCP-TRIM throughout all the cases. To illustrate the revenue from TCP-TRIM in controlling packet loss, we also record the number of dropped packets, as shown in Fig. 9(c). Overall, the drop amount of TCP becomes larger as the number of concurrent PTs increases, while TCP-TRIM does not experience packet loss and TCP timeout at all.

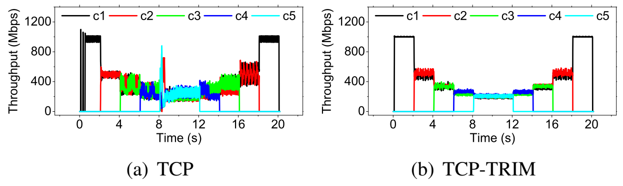
Goodput of the bottleneck link: from Fig. 9(d), we observe that TCP-TRIM achieves higher goodput than that of TCP for almost all the cases, and bottleneck link utilization is nearly 98% as well. Meanwhile, the almost full bottleneck link utilization in turn testifies the analysis of K configuration described in Section III.B.



(a) Switch queue with 5 LPTs (b) Average queue length
(c) Dropped packet number (d) Goodput of the bottleneck link

Fig. 9. TCP-TRIM properties.

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 100 packet buffer. 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. 10.



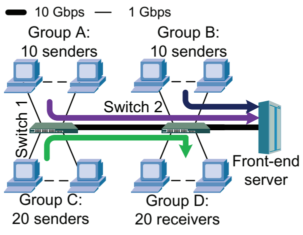
(a) TCP (b) TCP-TRIM

Fig. 10. Convergence test ("c1, c2..." means "connection 1, connection 2...").

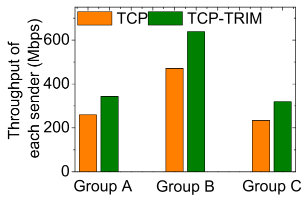
From the results, we can observe that TCP-TRIM benefits a lot from its better queue control. Altering the intensity of input traffic does not greatly disorganize the bandwidth share of TCP-TRIM. Consequently, each of the five connections converges to their fair share quickly. For TCP, although their throughputs are approximately fair on average, the convergence process shows large variation.

Multi-hop networks: to evaluate TCP-TRIM's performance in a multi-hop, multi-bottleneck environment, we build up a simulated network whose topology is shown in Fig. 11(a). Both group A and B have 10 senders, and they all send LPTs to the front-end server. Meanwhile, each sender in group C also sends a LPT to an receiver selected from the 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 other links are with 1 Gbps bandwidth. The PTs from group A go through all the bottlenecks.

The results are shown in Fig. 11(b). With TCP-TRIM, each server in group A obtains 342.7 Mbps and group B obtains 638 Mbps throughput, while each group C sender gets about 318 Mbps on average. On the contrary, TCP performs a little worse (259 Mbps, 471 Mbps, and 233 Mbps, respectively), frequent buffer overflows cause plenty of drops and TCP timeouts for some of the TCP connections, which finally leads to the lower throughputs.



(a) Multi-hop topology



(b) Throughput of each sender

Fig. 11. Multi-hop scenario.

C. Comparing with the State-of-art Data Center Protocols

To further understand the performance of TCP-TRIM in typical data center scenario, we set up the popular fat-tree topology network [14], and select the recently proposed well-known transport protocols in data center network, DCTCP and L^{2}DCT, to make the comprehensive performance comparison. The parameter settings in DCTCP and L^{2}DCT are in line with [3] and [16] respectively.

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 use ECN, and also weights

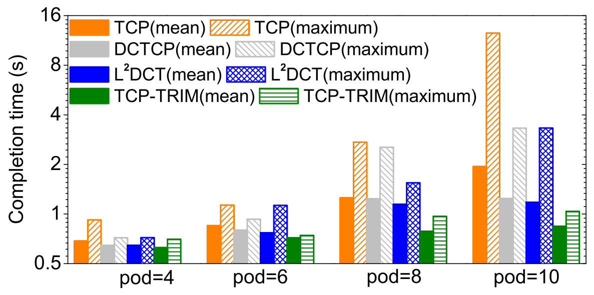


Fig. 12. Mean and maximum completion times in 10 Gbps fat-tree network.

TABLE I
THE NUMBER OF TIMEOUTS IN EACH PROTOCOL.

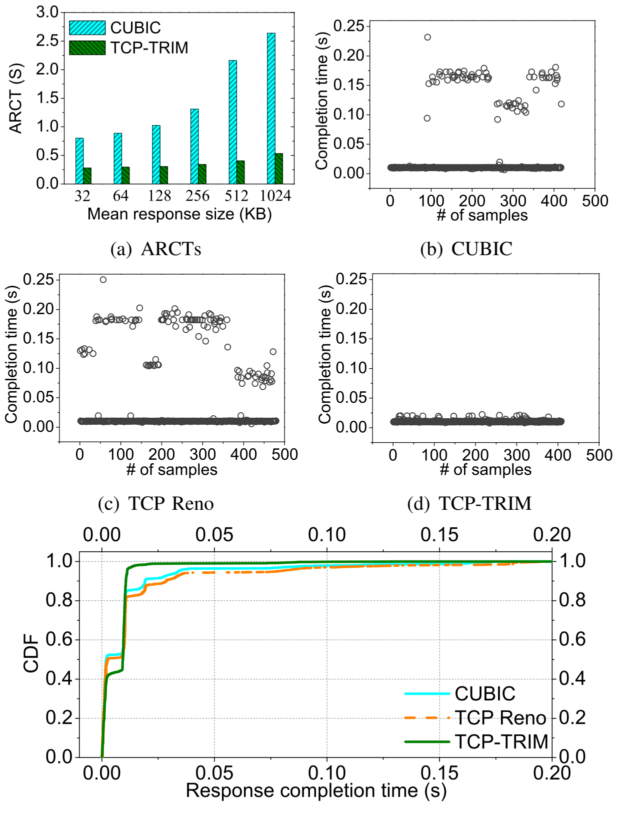
Pod number	TCP	DCTCP	L^2DCT	TCP-TRIM
4	13	9	9	8
6	85	75	71	39
8	452	440	274	141
10	1738	859	493	285

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 revenue 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 I, TCP still experiences the largest number of timeout events, and is followed by DCTCP and L^2DCT. TCP-TRIM always gets the least timeouts, especially when pod number is 10, the improved ratio comparing to TCP is up to 80%.

D. Real Implementation

In this section, we use several DELL OptiPlex 3010 Desktop machines, which act as the back-end servers, to test the performance of TCP-TRIM. These machines connect to the front-end server (CPU: Intel XEON E5-2650, MEMORY: 24G) via a switch with 100 Mbps links and 1 Gbps links, respectively. The kernel patch for supporting TCP-TRIM is pre-installed into all the servers. In the first real implementation, 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. 13(a), we observe that the ARCTs in CUBIC and TCP-TRIM become larger as the mean response size increases. By contrast, however, the increasing trend of ARCTs in TCP-TRIM is more gentle, and with the help of TCP-TRIM, the response transfer finishes more quickly as well. TCP-TRIM brings revenues to all the cases in different degree. Furthermore, the larger the response, the larger the revenue.



(a) ARCTs
(b) CUBIC
(c) TCP Reno
(d) TCP-TRIM
(e) The CDF of response completion time

Fig. 13. The results of real testbed.

Next, we buildup a simple web service scenario, in which 4 DELL machines 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 CUBIC, TCP Reno and TCP-TRIM, and record the completion time of each response. Then, we pick out the completion time samples of responses whose sizes are from 64 KB to 256 KB, and show them in Fig. 13(b), Fig. 13(c), and Fig. 13(d). It is clear that all the samples in TCP-TRIM never exceed 25 ms, while in the other two protocols, quite a few samples are higher than 50 ms, and some of them even reach to 250 ms. This indicates TCP-TRIM could get smaller ARCT. For further testifying this observation, in Fig. 13(e), we give the distribution of the completion times of all the responses for each protocol. Again, since nearly 99% of the response completion times is below 25 ms, TCP-TRIM performs the best, thus bringing noticeable improvements in the reductions of ARCT and tail latency.

V. Related work

The enhanced TCP protocols can be classified into two categories: pure end-to-end ones and explicit feedback based ones. The explicit feedback based protocols can provide more accurate network congestion information, such as [3], [15], [16], and [17] etc., while at the same time facing the applicability problem. On the other hand, the popular end-to-end policies in diverse network environments can also be divided into two categories: delay-based congestion control [20], [21], [22], and [23] and packet loss-based congestion control. The latter performs by considering packet loss only, while the former attempts to avoid congestion based on variations in RTT.

In the literature of data center transport mechanisms, many recent schemes try to deal with the highly concurrent communication. 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. However, the practicability is not satisfactory because its implementation relies on whether the switch supports ECN. D^2TCP [15] is proposed based on DCTCP. It considers both the congestion control and deadline requirements by elegantly adjusting the extent of window decreasing. When congestion occurs, far-deadline flows release some bandwidth to near-deadline flows, hence more flows can meet their deadlines. Nonetheless, just like DCTCP, the ECN machinery is also necessary. L^2DCT [16] still follows the properties of DCTCP in concurrency control while introduces the Least Attained Service (LAS) scheduling at the sender. Besides, in order to solve the incast problem [18], [19], J. Zhang et al. proposes GIP, which starts the transfer of each stripe unit in a TCP flow with window size 2 to minimize packet loss, and also redundantly transmit the last packet of each stripe unit to further alleviate TCP timeout [13]. However, we show that the bottleneck link would be underutilized if the network capacity is actually large enough to accommodate each flow to start with a large congestion window.

As to the up-to-date delay-based schemes in data center network, both [22] and [23] 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 methods that exploit latency-based congestion feedback to keep the delay low while delivering high throughput.

Overall, to our best knowledge, none of the above works specially focus on handling the improper congestion window evolution on concurrent HTTP connections, which is just 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.

[2] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken, The Nature of Datacenter Traffic: Measurements & Analysis, *in Proc. IMC*, 2009.

[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.

[4] 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.

[5] 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.

[6] C. Joachim, "HTTP/TCP connection and flow characteristics," *Performance Evaluation*, vol. 42, no. 2, pp. 149-162, 2000.

[7] 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.

[8] Y. Chen, R. Mahajan, B. Sridharan, and Z. Zhang, A Provider-side View of Web Search Response Time, *in Proc. ACM SIGCOMM*, 2013.

[9] J. J. Lee and M. Gupta, A new traffic model for current user web browsing behavior, *in Proc. Intel corporation*, 2007.

[10] H. Choi and J. O. Limb, A behavioral model of web traffic, *in Proc. IEEE ICNP*, 1999.

[11] The Network Simulator–ns-2, http://www.isi.edu/nsnam/ns, 2014.

[12] 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.

[13] J. Zhang, F. Ren, L. Tang and C. Lin, Taming TCP Incast Throughput Collapse in Data Center Networks, *in Proc. IEEE ICNP*, 2013.

[14] 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.

[15] B. Vamanan, J. Hasan, and T. N. Vijaykumar, Deadline-Aware Datacenter TCP (D^2TCP), *in Proc. ACM SIGCOMM*, 2012,

[16] 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.

[17] 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, June. 2015.

[18] 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.

[19] 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.

[20] R. Jain, "A delay-based approach for congestion avoidance in interconnected heterogeneous computer networks," *ACM Computer Communication Review*, vol. 19, no. 5, pp. 56-71, October. 1989.

[21] L. Brakmo, S. OMalley, and L. Peterson, TCP Vegas: New techniques for congestion detection and avoidance, *in Proc. ACM SIGCOMM*, 1994.

[22] 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.

[23] C Lee, C Park, K Jang, S Moon, and D Han, Accurate Latency-based Congestion Feedback for Datacenters, *in Proc. USENIX ATC*, 2015.