Enhancing TCP via Hysteresis Switching: Theoretical Analysis and Empirical Evaluation

Ahmed M. Abdelmoniem, *Member, IEEE, ACM*, and Brahim Bensaou, *Senior Member, IEEE, Member, ACM*

***Abstract*— In this paper we study the relationship between the TCP packet loss cycle and the performance of time-sensitive traffic in data centers. Using real traffic measurements and analysis, we find that such loss cycles are not long enough to enable most partition-aggregate time-sensitive TCP applications to recover their packet losses via the TCP 3-dup ACKs mechanism. As a result, the Timeout (RTO) mechanism is frequently triggered, leading to the expansion of the flow completion times (FCT) of such applications by orders of magnitude. Hence, we seek an alternative method that does not change the virtual machines and that can effectively expand the loss cycle duration to enable short flows to finish their transfer without incurring the cost of the RTO. To this end, we propose a novel TCP-AQM mechanism that alternates between a slow constant bitrate (CBR) mode and a fast TCP rate via hysteresis switching to expand the loss cycle. We prove the stability of the proposed TCP-AQM via a control theoretic model, then evaluate its performance gains via small and large scale NS2 simulation and by real FPGA implementation of a prototype on the NetFPGA platform. The results show considerable improvements in FCT distribution and reduction of missed deadlines in simulation and real experiments.**

***Index Terms*— Data center, congestion control, hysteresis.**

I. Introduction

TCP is the most widely used transport protocol in cloud applications. At the base, TCP is a distributed end-to-end protocol that relies on a collection of algorithms to achieve reliability. Most of these algorithms were not part of the early incarnation of TCP; they were added gradually to keep up with the changes that affected the Internet and the traffic over the decades. As a result, most TCP implementations found in today's operating systems are by default fine-tuned to operate efficiently in high-delay environments, such as the Internet. In particular, one algorithm that plays a key role in defining the performance of TCP applications is the congestion control (CC) mechanism. Over time, numerous variants of the TCP CC mechanism have seen the light. Most are designed to meet the ever evolving design goals and operational requirements of new operating environments [1], [2], [3], such as lossy wireless links e.g., TCP Westwood [1]; high-speed long-distance networks e.g., loss-based Cubic TCP [2] or delay-based Fast TCP [3], and so on.

Manuscript received 24 November 2021; revised 11 July 2022 and 24 November 2022; accepted 26 February 2023; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor D. Malone. Date of publication 7 April 2023; date of current version 19 December 2023. This work was supported in part by the Hong Kong Research Grant Council (RGC) under Grant GRF16209922. *(Corresponding author: Ahmed M. Abdelmoniem.)*

Ahmed M. Abdelmoniem is with the School of EECS, Queen Mary University of London, E1 4NS London, U.K., and also with the CS Department, FCI, Assiut University, Asyut 71515, Egypt (e-mail: ahmed.sayed@qmul.ac.uk).

Brahim Bensaou is with the CSE Department, The Hong Kong University of Science and Technology, Hong Kong (e-mail: brahim@cse.ust.hk).

Digital Object Identifier 10.1109/TNET.2023.3262564

In the same spirit, small short-lived flows have recently been observed to experience unnecessarily long FCT in data centers with vanilla-TCP; consequently, several new variants of TCP CC mechanisms have been proposed for data center networks (e.g., DCTCP [4], [5], TIMELY [6]). In contrast, other studies simply identified the sources of performance degradation in data centers with vanilla-TCP and proposed tuning the existing CC mechanisms' parameters to match the characteristics of data center networks (e.g., reducing the initial congestion window to cope with the small switch buffers [7] or scaling down the minimum RTO to match the small round-trip time (RTT) inside data centers [8]). All these approaches have been shown to yield some performance improvements, and some of them are already in use in production data centers. Nevertheless, in public multi-tenanted data centers where the virtual machines (VMs) are controlled by the tenants, it is not possible to modify or replace the TCP CC or to tweak the drivers inside the VM. So we argue that these solutions are only applicable to private data centers where the operator has control over both ends of the internal TCP connections. In addition, several works have investigated switch-based controllers to improve the FCT of short-lived flows. For example, pFabirc [9] and PIAS [10] leverage priority queuing in the switches to segregate and serve short-lived flows in priority. These mechanisms also apply exclusively to privately owned data centers because they require modifications of the TCP stack (e.g., PIAS relies on DCTCP, inheriting its drawbacks, and pFabric relies on a modified version of TCP).

In public multi-tenanted data centers, the tenants share a common physical infrastructure to run their applications on VMs. The tenants can implement and deploy their preferred operating system and thus version of TCP and TCP CC mechanism. To tackle this issue, a few approaches have been proposed in the literature. First, the public data center operator can statically apportion the network bandwidth among the tenants, giving each of them a fixed allocation with guaranteed performance bounds [11], [12]. Although effective, this technique, would not benefit from the statistical multiplexing, because of the high burstiness of TCP traffic, which results in an ineffectively used admissible region. Another approach suggests to modify all the switches in the data center to ensure small buffer occupancies at each switch. This can be achieved by using separate weighted queues and/or applying various marking thresholds within the same queue [13], [14]. Typically, each source algorithm requires a certain weight/threshold to fully utilize the bandwidth. That is why such schemes are not scalable, may lead to starvation and are hard to deploy due to the increasing number of different CC algorithms employed by the tenants. To enable true deployment potential in heterogeneous TCP environments

without modifying TCP, in this paper, we adopt a switch-based approach.

However, to make our scheme agnostic to the nature of the CC mechanism employed by the tenant, we rely on a universally adopted TCP mechanisms to convey congestion signals to the sources. The sources become simple flow rate enforcement entities while the switches covey the rates to the sources during congestion. This approach allows the operators to innovate freely in the switches and update them without worrying about the TCP variants chosen by the tenants[1]

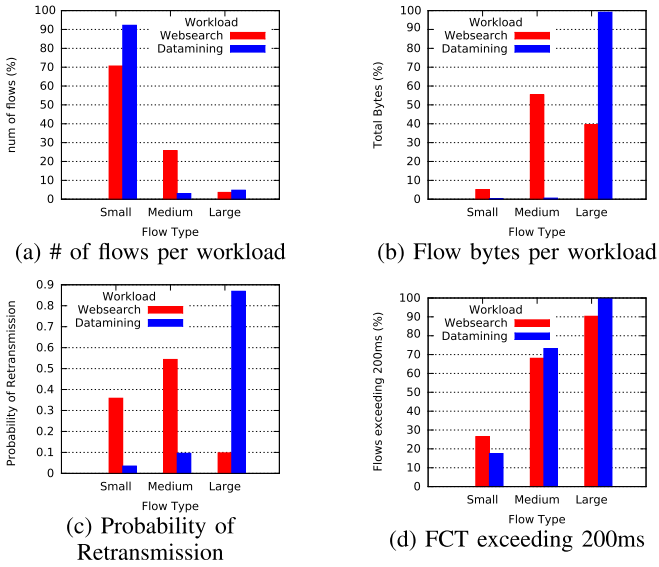
In the remainder, we first report our empirical results on the impact of TCP RTO on the performance of the traffic flows and then show the relation between the RTO and the TCP loss cycles in Section II. The proposed solution, is modelled, and some design and practical aspects are discussed in Section III. Then, in Section V, we discuss our implementation details and show experimental results from a real deployment in a small-scale testbed. We introduce important related works in Section VI. Finally, we conclude the paper in VII.

II. Background

Short-lived flows in data centers face the challenge of operating in small-buffered environments when off-the-shelf TCP uses inadequate, Internet-targeted mechanisms and parameters such as large initial congestion window, long minimum RTO and/or exponential window growth in slow-start. This combination of hardware and TCP configuration frequently leads to multiple timeout events for small flows, which inflates the FCT by several orders of magnitude. In particular, when the number (N) of such flows increases, in the presence of small buffers, synchronized losses (the so-called TCP incast congestion) occur. Knowing that the loss probability grows linearly with N [16], the flow synchronization and the excessive losses are known to lead to throughput-collapse for small-flows in data centers. To illustrate this, assume the link capacity C is shared equally among the N flows and let F be the flow size, τ be the mean RTT and x be the nominal number of RTTs necessary to complete the transfer. Then, the optimal throughput (ρ^*) can be expressed as: $\rho^* = F \big/ (x\tau + \frac{NF}{C})$. However, in practice, when TCP incast congestion involving N flows results in throughput-collapse, some flows experience, say, n timeout events and have to recover via RTO each time. Then, the transfer time becomes ($n\times$RTO + the typical transfer time) and the collapsed throughput (ρ^-) becomes: $\rho^- \sim F \big/ (n\text{RTO} + x\tau + \frac{NF}{C})$.

In data centers, the typical RTT is around 100μs, while existing TCP implementations impose a minimum RTO of about 200ms and above. For large long-lived flows, x is large and n is typically small, hence $nRTO$ and $x\tau$ in the aforementioned expression of ρ^- are comparable. In contrast, for small flows, that only last a few RTTs, $nRTO$ can be at least several orders of magnitude larger than $x\tau$, even for $n = 1$ [8], [17]. As a consequence, if a small flow experiences a loss that cannot be recovered by 3-dup ACKs (called hereafter a Non-Recoverable Loss or NRL), then it has a high chance of missing for example the service level agreement on the deadlines (e.g., ≈100ms). So, to improve the performance of small flows, it is necessary to curb or delay NRLs.

[1]A preliminary version of this work appeared in IEEE INFOCOM [15]. The source code is publicly available at https://github.com/ahmedcs/HSCC



(a) # of flows per workload (b) Flow bytes per workload (c) Probability of Retransmission (d) FCT exceeding 200ms

Fig. 1. An empirical study to characterize TCP flows and evaluate the effects of the RTO on the FCT in both the websearch and DataMining workloads.

A. Empirical Study of The Effects of RTO on the FCT

To support this analysis, we conduct experiments in a small-scale testbed to study the frequency of timeouts in high-bandwidth low-delay environments. We reproduce traffic workloads found in public and private data centers, by building a custom TCP traffic generator. The generator establishes TCP connections to mimic flows with sizes and inter-arrival time distributions drawn from various realistic workloads (e.g., WebSearch [4] and DataMining [18], as well as others [19], [20]). To conduct the experiment, we instrument the hosts with probing functions [21] similar to the procedure in [22]. We run a total of 7000 flows categorized into small flows ($\leq 1MB$), medium flows ($1 - 10MB$) and large ones ($\geq 10MB$).

Fig. 1a shows the ratio of flows generated from each size-category while Fig. 1b shows the ratio of network bytes generated from each size category. We observe that, in both workloads, most flows are small. But, in the WebSearch workload, data bytes are distributed almost uniformly over the three categories, In the DataMining workload, most of the traffic is produced by large flows (i.e., these flows tend to be quite large in size). Fig. 1c shows the probability of retransmission for different flow types observed in each workload. It suggests that retransmission is highly likely for all flow types in both workloads. Noticeably, in the WebSearch workload, the number of RTOs of small flows is ($\approx 35\%$). Fig. 1d shows the ratio of flows exceeding the FCT of 200ms for small flows to be high (i.e., $\approx 28\%, \approx 18\%$) for each workload, respectively.

From our measurement study, we find that flows would roughly experience on average 1/2 RTO in WebSearch (2 in DataMining). Which translates into adding another 100ms (400ms) or more to the FCT, i.e., more than 66% (366%) the ideal FCT for WebSearch (DataMining) workloads, respectively. This demonstrates the devastating effect

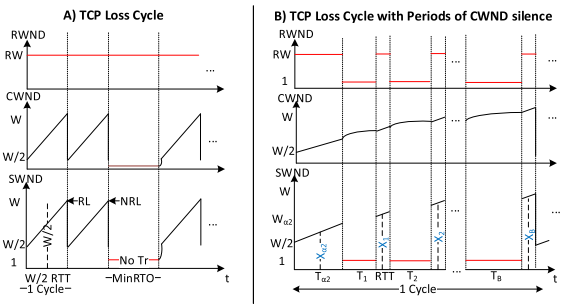


Fig. 2. A TCP loss cycle showing differences between the progress of TCP with DropTail vs. Hysteresis Switching. The hysteresis helps prolong TCP cycle allowing for more data transfer before experiencing a loss at the end.

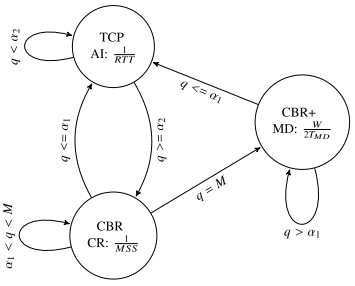


Fig. 3. Finite state machine of TCP with hysteresis control.

that RTOs could have on small flows, which agrees with the findings of previous studies [4], [8], [23], [24], [25], [26]. The measurements also strongly suggest that RTO frequency is non-negligible in data centers which suggests that the TCP loss cycle is so short that the flows in question do not have a sufficient share of the pipe bandwidth to inject a sufficiently large number of packets in flight to illicit 3-dup ACKs recovery.

III. The Proposed Methodology

In this part, we explore the anatomy of the problems stated in the previous sections and discuss the proposed solution.

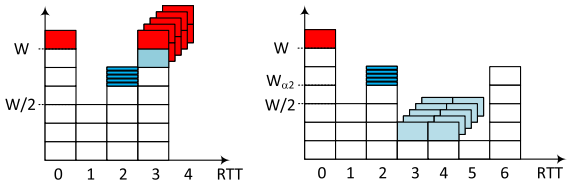
In congestion avoidance (CA) mode, TCP CC uses an additive-increase multiplicative-decrease (AIMD) algorithm which results in a congestion window ***Cwnd*** that evolves in a typical periodic sawtooth behavior as shown in Fig. 2(A). AIMD ensures that ***Cwnd*** attains all the values between the maximum ***Cwnd***(w), at which congestion occurs, and its minimum value (or equilibrium point $w/2$ [16]). This represents one loss cycle. The figure also shows that the flow control receiver window ***Rwnd*** is quasi constant, because receivers often allocate large enough receive buffers. The sending window ***Swnd*** $=$ min(***Rwnd***, ***Cwnd***) is also shown. In a typical scenario, because of the small RTT and frequent losses in data centers, ***Rwnd***$\gg$1; in addition, the throughput of TCP with AIMD can be shown to be inversely proportional to the square root of the loss event probability [16]. Therefore, due to frequent loss events in data centers, the loss cycle is very short, and thus ***Rwnd***$\gg$***Cwnd***. Thus ***Swnd*** almost always takes the value of ***Cwnd***, with ***Rwnd*** being ineffective in such case. This problem is encountered with window-based TCP (e.g., Reno, Cubic, DCTCP) that rely on losses as congestion signals to adjust their sending rates.

Because of frequent loss events, small flows end up by not having enough time to build a long enough flight of packets to trigger 3-dup ACKs for loss recovery, which leads to frequent timeouts. To increase TCP throughput one needs to curb such non-recoverable losses by stretching the loss cycle to enable small flows to build their data flight size to values that allow for 3-dup ACKs or to complete the transfer before a loss occurs. In this spirit, we propose to use hysteresis to control the traffic flow into the router's buffer by setting two thresholds: a low threshold α_1, below which the sources can start to send data according to TCP CC ***Cwnd***, and a high threshold α_2 above which all the sources are throttled to a small constant bit-rate (CBR) (e.g., 1MSS/RTT) that would enable the congestion in the router buffer to recede. Such CBR mode would remain in effect until the backlog clears and the buffer occupancy drops below α_1 threshold. Such behavior is better described by the finite state machine in Fig. 3. Assuming slow start is transient, starting in AIMD mode in the state named 'TCP', the source rate alternates between using ***Cwnd*** with additive increase in the state TCP and a constant bit-rate (CR) of 1MSS/RTT in the state 'CBR', as long as the queue does not overflow the buffer (of size M). Once a loss occurs, not only the CR is imposed on the sources, but also TCP's multiplicative decrease (MD) is imposed on ***Cwnd*** , as depicted in the state 'CBR+'.

To build such a controller without changing TCP in the VM, one can imagine an oracle that can suppress the use of ***Cwnd*** during the CBR and CBR+ states and replaces it with a constant rate. Since the source sending rate ***Swnd*** $=$ min(***Rwnd***, ***Cwnd***), and ***Rwnd*** has little effect on the source rate in data centers, such an oracle could simply be the switch itself that is *i)*aware of the instantaneous buffer backlog, and *ii)*could impose the constant rate on the sources by simply rewriting the CR value in the ***Rwnd*** field of the ACK packet headers, as long as the state is in CBR or CBR+.

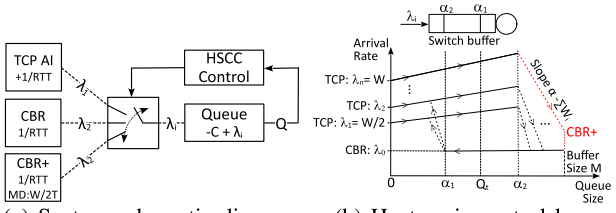
Under such a controller, losses are delayed; and the loss cycle is stretched as illustrated in Fig. 2(B) that shows ***Cwnd***, ***Rwnd*** and ***Swnd*** in one loss cycle. In each cycle, ***Cwnd*** becomes inactive for a certain time and ***Rwnd*** takes over the control of ***Swnd***. During this period, ***Rwnd*** is set by the switch to a small rate until congestion recedes and ***Cwnd*** can be reactivated again with its suspended value. During the time where ***Rwnd*** controls the sending rate, ***Cwnd*** continues to increase according to the AIMD algorithm as shown in the figure, which ensures that TCP fairness among flows is still maintained. It is easy to demonstrate analytically that this approach indeed improves the throughput of TCP. Fig. 2(B) describes the expected TCP behavior with stretched loss cycle:

1) TCP starts increasing ***Cwnd*** in the AI mode until the window reaches w_{α_2} (which is triggered by the queue reaching α_2 at the bottleneck switch) based on signals coming from the switch.
2) When ***Cwnd*** reaches the threshold, the state switches to the slow CBR mode which enforces sending at rate $1\ MSS/RTT$ via ***Rwnd*** until an implicit signal comes to signal that congestion has receded, which allows the source to resume using its previous ***Cwnd*** .
3) Afterwards, TCP will operate in AI mode for at least one RTT before switching back to the slow CBR mode.
4) This cycle of switching back and forth between AI and slow CBR continues for a few more RTTs until ***Cwnd***



(a) TCP-Droptail (b) TCP-HSCC

Fig. 4. Toy scenario with incast: TCP with DropTail vs. HSCC.



(a) System schematic diagram (b) Hysteresis control law

Fig. 5. (a) HSCC system components, the feedback loop and the hysteresis law which switches between TCP, CBR and CBR+ traffic sources. (b) The control law of HSCC obeys a counter-clockwise hysteresis to switch between states based on queue occupancy.

reaches or exceeds the maximum congestion window w that leads to congestion.

5) Then a packet loss occurs and triggers MD and CBR, after which a new stretched loss cycle starts. ***Cwnd*** that was cut in half resumes the rate control.

Fig. 4 shows a toy scenario where an ongoing steady state TCP flow shares the network with 5 new incoming incast flows. In deep blue colour are the TCP-SYN packets to open the 5 incast connections (we assume each of them has two segments to transmit and the initial window is 2 MSS). In Fig. 4a, at RTT 2, 5 SYN segments are acknowledged leading to 5 new connections starting to send two MSS each in RTT 3. In the presence of the existing TCP flow continuing with CA, this leads to only one segment (in light blue) being successfully received and 9 segments being lost, inducing a timeout for all 5 flows. That is, at RTT 2, the window of the ongoing flow was at $w-2$ and by the time the incast flows start sending data at RTT 3, the ongoing flow has already increased its window to $w-1$ leaving room for one extra segment only. In stark contrast, Fig. 4b shows that at RTT 2, the hysteresis switch would be active and the current sending window of all flows is set to 1 MSS. Therefore, from RTT 3 onward, and as long as the total window is above w_{α_2}, each flow including the new flows and the ongoing flow, would be allowed to send 1 MSS per RTT only. We can clearly see that all 6 flows are able to continue sending data albeit at reduced rates. After two RTTs, the five incast flows finish their transmission and the original flow can resume using its suspended congestion window immediately after. The performance gains in terms of FCT for incast flows and network throughput are obvious.

A. System Design and Modeling

HSCC system control loop is depicted in Fig. 5a. The system consists of four main components: three data sources, and the queue with a hysteresis controller that switches among the three data sources. The sources comprise a TCP Additive Increase source with 1 MSS/RTT increase rate, a CBR source sending at a constant rate of 1 MSS/RTT, and a CBR+ source

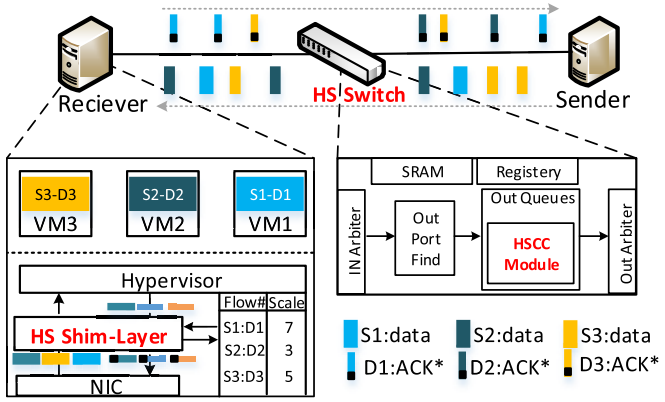


Fig. 6. HSCC system real deployment components.

that combines the CBR source and a Multiplicative Decrease applied to the congestion window. Fig. 5b shows the switching control law used by the HSCC switch as a counter-clockwise hysteresis system. Starting with a TCP source with rate λ_1, the switching to CBR with rate λ_0 first happens when the high threshold α_2 is crossed in the buffer. Such state continues with a constant rate λ_0 until the lower threshold α_1 is crossed. The system switches back to TCP state but this time with rate λ_2[2] and so on until the system enters the TCP state with a congestion window that leads to a packet loss. In this case the system switches to CBR+ which is similar to CBR except that ***Cwnd*** is also cut by half. This ends one cycle.

Fig.3 shows the transition diagram of TCP-HSCC system states. It is clear that there is a singularity in the congestion window value (when it is cut by half) one RTT from the loss event. These events are caused by the queue length exceeding the buffer size M as shown in the diagram. However, in loss events, the HSCC control law is already operating with CBR mode, because, to reach full buffer occupancy, the buffer must cross the high threshold first. It is clear from the transition diagram that the system is operating with CBR until the queue falls back to the low threshold which starts the AIMD mode again. This means there is a time-span in which the new congestion window $\frac{w}{2}$ is inactive (i.e., the duration for the switch to drain the queue back to the low threshold).

Fig. 6 shows the HSCC system practical components and operations. It consists of the HSCC switch module that performs the Hysteresis switching and an end-host module residing under the hypervisor (HS Shim layer) whose role is to translate window scaling option. First, at connection-setup, flows are hashed into a hash-table with the flow's 4-tuples (source and dest. IP address, and source and dest. port number) used as the key and the receiver window ***Rwnd*** scale factor used as the value. Flow entries are cleared from the table when a connection is closed (i.e., FIN is received). The module writes the scale factor for all outgoing ACK packets in the 4-bit reserved field of TCP headers.[3] The used bits for window scaling are cleared after their usage by the HSCC switch to avoid packets being dropped by the destination due to invalid TCP checksum value. This saves the

[2]Note that despite the sending rate is controlled by the ***Rwnd***, the returning ACKs in the VM continue to increase ***Cwnd***, albeit with very small fractions.

[3]Instead, 4-bits of the receive window field could be used to encode the window scaling and the remaining 12 bits used for the actual window values.

need to recalculate the checksum at both the end-host and the switch. As shown in Fig. 6, the module resides right above the NIC driver for a non-virtualized setup and right below the hypervisor to support VMs in cloud data centers. Hence, this does not impact the network stack implementation of the host or guest OS, making it readily deployable in production data centers. The end-host module tracks the scaling factor used by local communication end-points and explicitly append this information only to outgoing ACKs of the corresponding flow. The switch module on the other hand monitors the queue and whenever it detects that the hysteresis (high) threshold has been crossed for one of the ports, it immediately switches to CBR mode and apply receiver window updates in the incoming ACKs of that port.

B. Practical Aspects of The System

Control Packets Loss: TCP packets are considered lost if the data or its corresponding ACK is lost and connections cannot be opened/closed if the SYN/FIN segments are lost. Hence, if ***Rwnd*** is set to small values such as 1 MSS, then an ACK segment loss can lead to timeout. To address this issue, HSCC may leave room in the buffer for any control packets (e.g., SYN, FIN, ACK and so on) to safeguard them from possible losses. In our design, we set another drop point σ on all switch queues below the size of the buffer: σ is set to reserve a small amount of buffer (e.g., ≈3%) for the different control packets like pure ACK packets. If ACKs are piggy-packed, then similar to [25], the switch could cut the payload and forward only the ACK. The evaluation of these tweaks is not covered in this paper.

Receive Window Scaling: HSCC relies on a scale factor to rescale the modified window written into TCP header of incoming ACKs. TCP specification [27] states that the three-byte scale option may only be sent in SYN segments by each TCP end-point to let its peer know what factor it uses for its own window value scaling. The scaling may be unnecessary for networks with Bandwidth-Delay Product (BDP) of 12.5KB (i.e., with NICs of 1 Gbps and RTT≈ 100μs). However, with the adoption of high speed links of 10 Gbps (i.e., BDP = 125KB), 40 Gbps (i.e., BDP = 500KB) and 100 Gbps (i.e., BDP = 1.25MB), the scaling factor becomes necessary to utilize the bandwidth effectively. This applies to cases when there are less than 2 (for 10Gbps), 8 (for 40Gbps) and 20 (for 100Gbps) active flows. Even though, the probability of having such small number of active flows per host in data centers are extremely small [4]. HSCC should be designed to handle window scaling via flow-level tracking. Since implementing this at the switch would result in a non-scalable system, we propose the light-weight end-host shim-layer to explicitly send scaling factor with outgoing ACKs. The shim-layer extracts and stores the advertised scaling factor (i.e., from the window scaling option) from outgoing SYN and SYN-ACK segments for each established TCP flow. The shim-layer encodes the scaling factor using 4 of the 8 reserved bits of the TCP header. Then, the switch uses this value to scale the new window properly and clears the used bits. In Ethernet networks, IP checksum is not checked by forwarding devices and checked at the receiver IP layer. So by clearing the used bits, computation of new IP checksum is not necessary both at the shim-layer and at the switch.

IV. System Modeling and Stability

In this section, we will study the stability of our controller using the standard fluid modeling and linearization methods. Fig. 3 shows the state transitions in TCP-HSCC where the control alternates between two systems namely AIMD of TCP and the slower CBR when HSCC is actively rewriting ***Rwnd***. Since the two states happen disjointedly, we model TCP-HSCC behavior, via two decoupled sets of differential equations that represent the dynamics of each system then combine them as a weighted sum. For this define $p_u \in [0, 1)$ as the ***Rwnd*** update probability (i.e., the average fraction of time when CBR is active). Define $p_l \in [0, 1)$ as the loss event probability triggered via 3-dup Acks. Similar to [28], [29], we assume a finite buffer capacity M, and that the difference in RTT among competing flows to be negligible, therefore the congestion windows and RTTs of the different flows converge respectively to $w(t)$ and $\tau(t)$. Assume also that the packet sizes are constant and the sources have a continuous supply of data, that is, short-lived flows are considered to be a temporal low-frequency disturbance/noise imposed on the system and are absorbed by the system dynamics. Then, let N represents the steady state number of flows sharing the buffer. The RTT for a packet with a bottleneck link capacity of C is $\tau_i(t) = T_c + T_t + T_p + \frac{q_i(t)}{C}$, where T_t is the transmission time, T_p is the propagation delay, T_c is the processing delays on the path and $\frac{q_i(t)}{C}$ is the queueing delay seen by flow i at time t. Let $t' = t - \tau(t)$, then the system dynamics can be stated as:

$$\frac{dw(t)}{dt} = \frac{1 - p_u(t')}{\tau(t)} + \frac{1}{w(t)}\frac{p_u(t')}{\tau(t)} - \frac{w(t)p_l(t')}{2}\frac{w(t')}{\tau(t')},$$
$$\frac{dq(t)}{dt} = Nw(t)\frac{(1 - p_u(t'))}{\tau(t)} + N\frac{p_u(t')}{\tau(t)} - C. \quad (1)$$

The first line in Eq.1 describes the congestion window dynamics and consists of three terms; the first term on its right-hand side indicates additive increase by 1 MSS per RTT when not in ***Rwnd*** update mode, which happens with probability $1-p_u(t')$, the second term indicates the CBR constant rate of 1 MSS when in CBR mode, which happens with probability $p_u(t')$, and the third term indicates multiplicative decrease when a loss event occurs. The second line in Eq.1 describes the bottleneck queue dynamics and consists of two terms; the first term represents the arrival data rate when the system is operating under TCP mode, and the second term represents the arrival data rate when the system is under the CBR mode. The derivations are a modified version of window and queue dynamics of TCP as in [28] and [29].

Proposition 1: The system described in (1) is stable

Proof: To show that the system is stable, it is sufficient to show that it is Hurwitz-stable after linearization [30].

The system state is defined by the pair $(w(t), q(t))$ and has two inputs (p_l, p_u). By definition of the FIFO queue, it is straightforward to notice that since $\alpha_2 \leq M$ then $p_u \geq p_l$ (i.e., the backlog in the buffer crosses several times α_2 before one packet loss occurs). So, without loss of generality, we can write $p_u = p$, $p_l = \kappa p$, where $\kappa \in [0, 1]$ is a positive scalar that is on average inversely proportional to the number of times the queue length exceeds the high threshold α_2 before a single buffer overflow occurs. Note that, after every time α_2 threshold

is crossed, the window increases by 1 MSS per flow when the TCP mode becomes active again. And so if N is given then κ can be calculated from the ratio of N, to the number of packets filling the queue between the high threshold α_2 and the full buffer size M (i.e., $\kappa = \frac{N}{(M-\alpha_2)}$). The operating point of the system is when the system dynamics comes to rest (i.e., the equilibrium point which is defined by the following parameters $(w_0, q_0, p_{u_0} = p_0, p_{l_0} = \kappa p_0)$. Given $\tau_0 = \frac{q_0}{C} + T$, the equilibrium point parameters p_0 and w_0 are as follows:

$$\begin{aligned} \frac{dw(t)}{dt} = 0 \quad &\rightarrow \quad P_0 = \left(\frac{\kappa w_0 \tau_0}{2} - \frac{1}{w_0} + 1\right)^{-1} \\ \frac{dq(t)}{dt} = 0 \quad &\rightarrow \quad w_0 = \frac{\frac{C\tau_0}{N} - p_0}{1 - p_0}, \end{aligned} \tag{2}$$

Linearization. we define the perturbed variables as $\delta w = w - w_0$, $\delta q = q - q_0$ and the perturbed system input $\delta p_u = p_u - p_{u_0}$, and denote by $\dot{w}(t) = dw(t)/dt$ and $\dot{q}(t) = dq(t)/dt$; then, the linearized system of equations around the equilibrium writes:

$$\begin{aligned} \delta\dot{w}(t) = &-\frac{p_0}{\tau_0}\left(\frac{1}{w_0\tau_0} + \frac{\kappa w_0}{2}\right)\delta w(t) - \frac{p_0}{C\tau_0^3}\delta q(t) \\ &- \frac{1}{\tau_0 P_0}\delta p(t - \tau_0), \\ \delta\dot{q}(t) = &(1 - p_0)\frac{N}{\tau_0}\delta w - \frac{1}{\tau_0}\delta q - \left(C - \frac{N}{\tau_0}\right)\delta p(t - \tau_0). \end{aligned} \tag{3}$$

Stability Analysis. Let $V_1 = \frac{2P_0}{\tau_0}\left(\frac{1}{w_0\tau_0} + \frac{\kappa w_0}{2}\right)$, $V_2 = \frac{P_0}{C\tau_0^3}$, $V_3 = (1 - P_0)\frac{N}{\tau_0}$, and $V_4 = \frac{1}{\tau_0}$: V_1, V_2, V_3, and V_4 are positive if $w_0 > 0$ and $P_0 > 0$. From (2), it is easy to see that $w_0 > 0$ if $C\tau_0 \geq N$ and $0 < P_0 < 1$. The condition $P_0 > 0$ means $\frac{1}{w_0} < 1 + \frac{\kappa w_0 \tau_0}{2}$ which is true if $w_0 \geq 1$. Similarly, the condition $P_0 < 1$ is true if $0 < \frac{\kappa w_0 \tau_0}{2} < 1$ and $w_0 \geq 1$. So the variables V_1, V_2, V_3 and V_4 are positive if $C\tau_0 \geq N, w_0 \geq 1$ and $0 < \kappa w_0 \tau_0 < 2$. Define A, X, b and y as:

$$\begin{aligned} A = \begin{bmatrix} -V_1 & -V_2 \\ V_3 & -V_4 \end{bmatrix} \quad , \quad x = \begin{bmatrix} \delta w \\ \delta q \end{bmatrix}, \\ b = \begin{bmatrix} -\frac{1}{\tau_0 P_0} \\ -(C - \frac{N}{\tau_0}) \end{bmatrix} \quad \text{and} \quad y = \delta p(t - \tau_0); \end{aligned} \tag{4}$$

then, (3) can be written in matrix form as: $\dot{x} = Ax + by$.

To prove that this system is stable, it is sufficient to show that A is a Hurwitz matrix, i.e., all its eigenvalues have negative real parts [30]. A has two eigenvalues:

$$\lambda_{1,2} = -\frac{V_1}{2} - \frac{V_4}{2} \pm \frac{\sqrt{V_1^2 - 2V_1V_4 + V_4^2 - 4V_2V_3}}{2}. \tag{5}$$

If $V_1^2 - 2V_1V_4 + V_4^2 - 4V_2V_3 <= 0$ then $\Re(\lambda_{1,2}) < 0$. Otherwise, if $V_1 + V_4 < \sqrt{V_1^2 - 2V_1V_4 + V_4^2 - 4V_2V_3}$, one of the eigenvalues may be positive; however, by simple algebraic manipulations of the inequality, we can show that for such case to happen $4V_1V_2 < 0$ which is impossible since V_1, V_2, V_3 and V_4 are all positive quantities; therefore the system is stable. ■

Sensitivity Analysis of The System: we conduct simulation experiments using WebSearch workload with various values of the thresholds $\alpha 1$ and $\alpha 2$ to assess the sensitivity of HSCC to their settings. The details on simulation settings, flow sizes, inter-arrival times and network loads are the same as in [15]. The results indicate that FCT is not affected at all by the choice of the parameters $\alpha 1$ and $\alpha 2$ and similar results are also observed for DataMining workload. This is not surprising because in all cases the system switches between TCP and low rate CBR but at slightly different times (i.e., switching time differences are in the sub microsecond scale) as well as the small size of the switch buffers. This means that our scheme is robust and the operator can deploy it without worrying about the right values for thresholds. However, we believe further testing and verification in real deployments are still necessary.

V. Implementation and Experiments

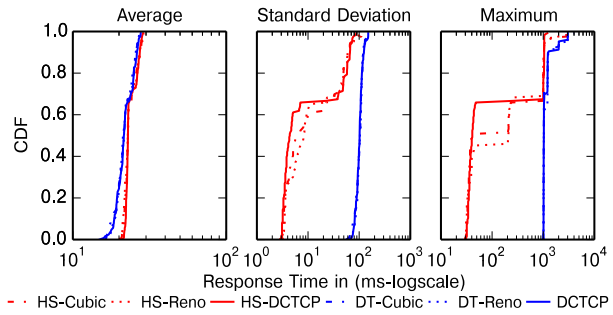
A simulation study of HSCC early design has been conducted in [15], therefore for brevity in this paper we only focus on investigating the performance of HSCC via a hardware prototype implemented with Verilog as a new switch design on the NetFPGA platform. The new switch is used to conduct a series of testbed experiments in a small data center to verify its potential benefits. We will first start the experiments via a synthetic micro-benchmark to understand the performance of HSCC, then we will run a variety of experiments using realistic traffic from our traffic generator to assess the performance of HSCC under realistic data center traffic.

Our testbed consists of 14 servers with 12 cores and 6 1Gbps Ethernet cards each connected via a 2-level FatTree topology. Four of the Ethernet cards in each server are reserved for building the testbed data center (the others are left for control and communication). Each card is bonded to two cores and is connected to a ToR switch allowing us to form 4 virtual racks. The four racks are interconnected via a 4 ports NetFPGA core switch on which we built the HSCC controller. By controlling the source and destination of the traffic, we can create bottleneck links as needed.

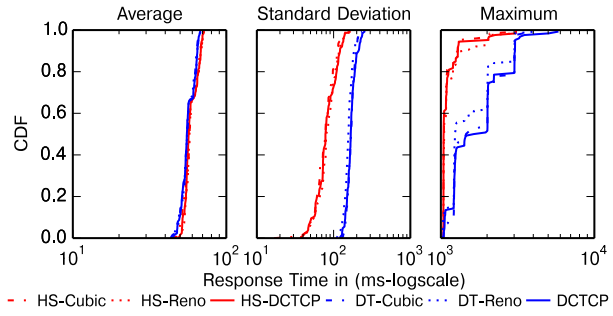
A. Experimental Results Benchmark Scenario

Incast Traffic without Background Workload: First, we run one mild and one heavy incast traffic scenarios where a large number of small flows transfer 11.5KB of data each. In both scenarios, 7 servers in rack 4, issue 1000 web requests to retrieve *"index.html"* webpage of size 11.5KB from the other 21 servers in rack 1, 2 and 3. In the mild load scenario, each requester uses 2 parallel TCP connections to finish the 1000 requests; hence, a total of 252 $((21 \times 7 - 21) * 2)$ synchronized requests are issued. In heavy load, the same number of requests are issued, however, each requester uses 5 parallel TCP connections instead of 2. This results in 630 incast flows (i.e., 126×5).

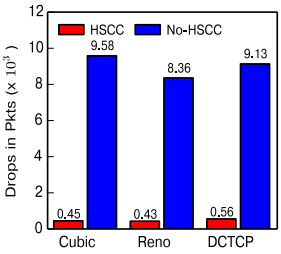
Experimental Results: Fig. 7 shows that HSCC achieves a significantly improved performance under both mild and heavy load scenarios. Even though, Fig. 7a shows that TCP-HSCC in the mild case achieves almost the same FCT on average compared to TCP-DropTail and DCTCP, it reduces, for more than 65% of the flows, the FCT standard deviation and maximum FCT by more than 1.5 orders-of-magnitude. This suggests that almost all flows (including tail-end ones) can finish before the end of their deadlines. In the heavy traffic case, Fig. 7b shows that the improvements are less significant compared to the mild case. This is because in heavy load case,



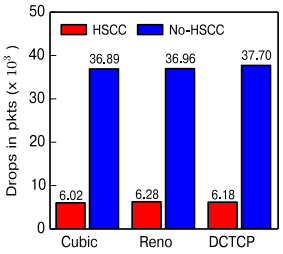
(a) FCT statistics. Mild traffic incast: 252 flows



(b) FCT statistics. Heavy traffic incast: 630 flows



(c) Mild traffic incast scenario



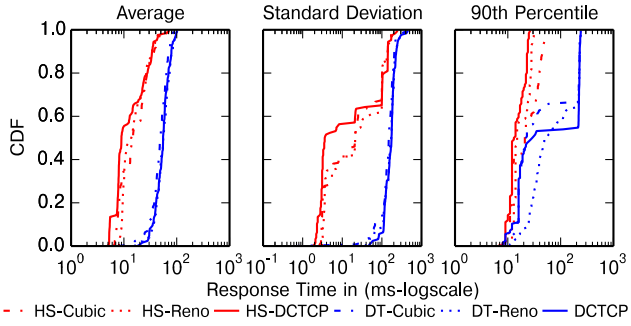
(d) Heavy traffic incast scenario

Fig. 7. Incast Scenario: FCT statistics and bottleneck link packet drops for TCP-HSCC, TCP-DropTail and DCTCP.

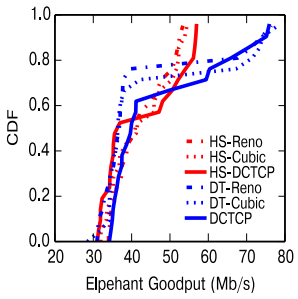
the CBR mode is always active and flows send at the minimum rate of 1 MSS per RTT, then the total rate is 920KB per RTT. The aggregate rate in this case is ≈ 3.2 times larger than the size of the bottleneck pipe (i.e., the switch buffer size plus the Bandwidth-Delay Product (BDP), or 287 KB). Fig. 8b and 8c show that HSCC can significantly decrease the in-network packet drop rate during incast events by $\approx 96\%$ in medium load and by $\approx 86\%$ in heavy load scenarios, respectively.

Mild Incast Traffic with Background Workload: now we examine HSCC performance when it is subjected to background long-lived flows. For this, we generate incast flows and let them compete for the same output port buffer with long-lived flows. To achieve this, we use iperf [31] to generate 21 long-lived flows set to send towards rack 4 continuously for 20s. As a result, the incast web traffic competes for the bottleneck bandwidth with each other as well as the new background traffic. A single incast epoch of Web requests is scheduled to run for 100 consecutive requests (i.e., each client requests a 1.15 MB file partitioned into 100 11.5KB chunks totalling $\approx$ 145 MB). This epoch is scheduled to start after the iperf flows have reached steady state (i.e., at the 10^{th} sec).

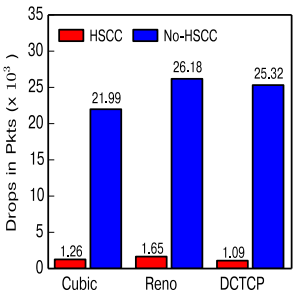
Experimental Results: Fig. 9a shows that, HSCC achieves FCT improvements for small flows while nearly not affecting the performance of long-lived flows. In addition, the FCT standard deviation is reduced by one order-of-magnitude



(a) FCT statistics for the 126 incast small flows running for 1 epoch



(b) Avg large flow throughput



(c) Total packet drops

Fig. 8. Mild Incast with background traffic: FCT statistics, bottleneck link packet drops and throughput.

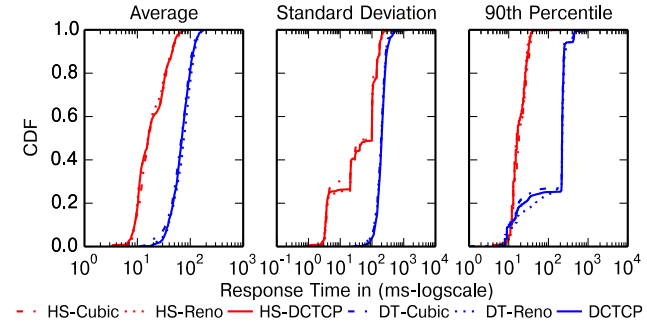
compared to TCP (Cubic, Reno) with DropTail and DCTCP. Finally, HSCC reduces the tail FCT (i.e., the 90^{th} percentile), by more than 1.5 order-of-magnitude. The improvements mean that small flows can finish quickly before their deadlines. Fig. 9c shows that long-lived flows are almost not affected by HSCC's intervention (throttling their rates during the short incast periods). Fig. 9b reveals that packet drop under HSCC is reduced considerably due to switching to the slow CBR during incast by throttling long-lived flows to avoid excessive packet losses for small flows.

Heavy Incast Traffic with Background Workload: We repeat the above experiment, increasing the frequency of shot-lived incast epochs to 9 times within the 20 second period (i.e., at the 2^{nd}, 4^{th}, ...and 18^{th} sec). In each epoch, the servers request a 1.15MB file partitioned into 100 11.5KB chunks (i.e., $\approx$145 MBytes per epoch and approximately 1.45 GBytes for all 9 epochs).

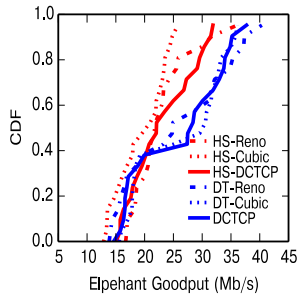
Experimental Results: Fig. 9a shows that HSCC scales well with higher incast frequencies even in presence of long-lived flows. The average FCT and standard deviation for small flows are significantly improved compared to TCP with DropTail and DCTCP. This can be attributed to the lower frequency of buffer overflows thanks to the expanded loss cycles in TCP-HSCC as shown in Fig.9c. The lower drop rate translates also into lower chance of experiencing timeout. Compared to the previous experiment, Fig. 9b shows that the long-term average throughput of 60% of the long-lived flows is reduced by roughly 5Mb/s. We believe that the system switches to CBR for long periods in heavy loads which enables fair utilization of the bandwidth and hence all flows (small and long) can progress equally and at the same rate.

B. Experimental Results With Realistic Workloads

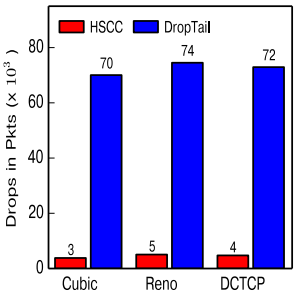
We use now the traffic generator described in Section II, to run experiments that involve realistic traffic from real data



(a) FCT statistics for the 126 mice flows running for 9 epochs



(b) Avg large flows throughput



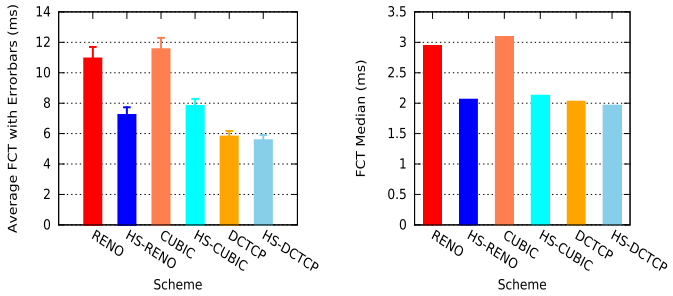
(c) Total packet drops

Fig. 9. Heavy incast with background traffic: FCT statistics, bottleneck link packet drops, and average throughput.

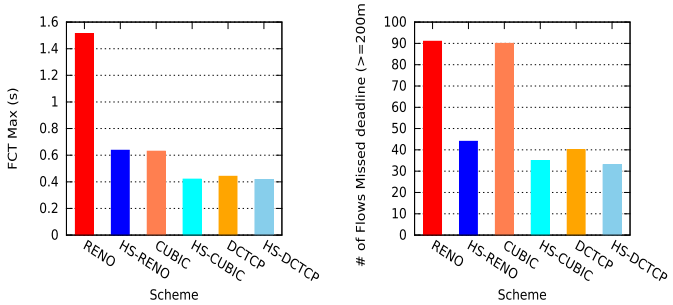
centers, namely, WebSearch and DataMining traffic workloads. In addition, in some experiments, we use iperf [31] to emulate some long-lived background traffic (e.g., VM migrations, backups and so on). We create a One-to-All scenario with and without long-lived background traffic. In this One-to-All experiment, clients from the VMs in one rack send requests randomly to any of all other servers in the cluster. In addition, in the experiment with background traffic, we run long-lived iperf flows in an all-to-all fashion to mimic sudden and persistent spikes in network load. Even though, we classify flows by size (i.e., $<= 100KB$ as small, $> 100KB \; and \; <= 10MB$ as medium and $>= 10MB$ as large), we focus more on the results of small flows which are the main target for HSCC.

A Scenario without Background Traffic: the traffic generator is set to randomly initiate 1000 requests per server per rack to randomly selected servers on one of the other racks. We report the performance in terms of average, median and maximum FCT and the number of flows that missed the 200ms deadline for small flows in Figures 10a, 10b, 10c and 10d for the WebSearch workload; and in Figures 11a, 11b,11c and 11d for the DataMining workload.

We can make the following observations: *i)* For WebSearch, HSCC improves the performance of small flows in both the average and maximum FCT as well as the number of missed deadlines for all TCP variants. For example, compared to Reno, Cubic and DCTCP, HSCC reduces the FCT of small flows by $\approx (34\%, 33\%, 5\%)$, $\approx (30\%, 32\%, 4\%)$ and $\approx (58\%, 34\%, 6\%)$ on the average, median and maximum, respectively. In addition, the number of missed deadlines is improved by $\approx (52\%, 62\%, 18\%)$ for Reno, Cubic and DCTCP, respectively. *ii)* For DataMining, the improvements are even more significant due to the predominance of small flows in this workload. For instance, compared to Reno,

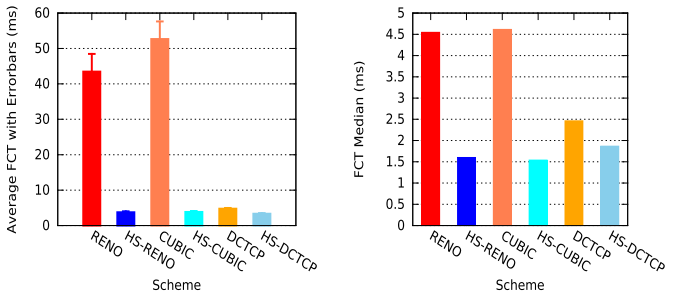


(a) Small: Average FCT (b) Small Flows: Median FCT

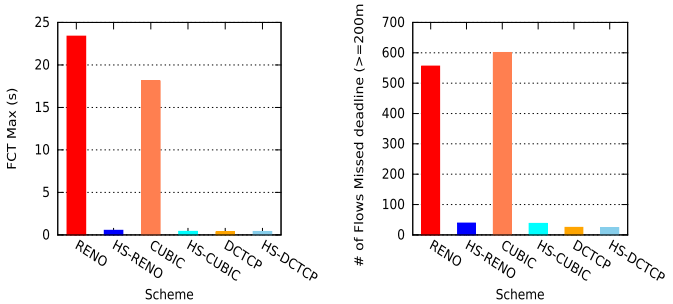


(c) Small Flows: Maximum FCT (d) Small: Missed deadlines

Fig. 10. One-to-All WebSearch; no-background traffic.



(a) Small: Average FCT (b) Small Flows: Median FCT

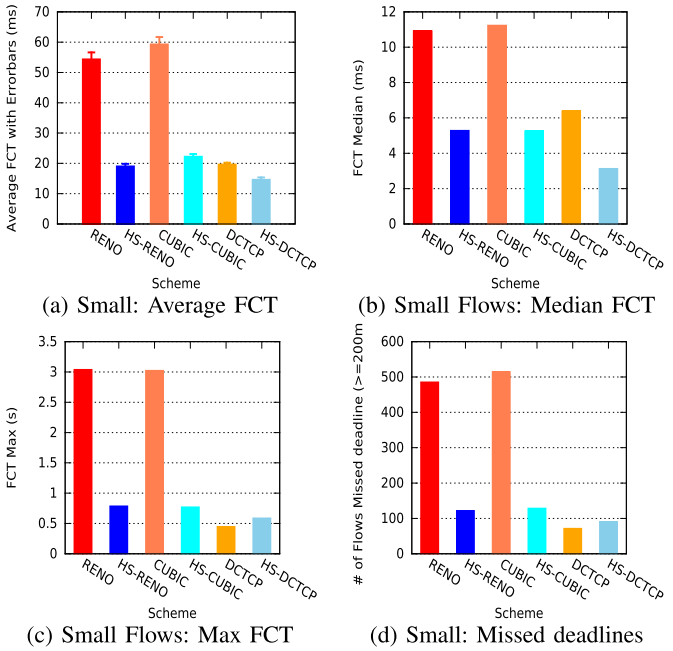


(c) Small Flows: Maximum FCT (d) Small: Missed deadlines

Fig. 11. One-to-All DataMining; no-background traffic.

Cubic and DCTCP, HSCC reduces the FCT of small flows by $\approx (92\%, 93\%, 30\%)$ on the average, $\approx (65\%, 67\%, 25\%)$ on the median and $\approx (98\%, 98\%, -\%)$ on the maximum, respectively. In addition, the number of missed deadlines is improved by $\approx (93\%, 94\%, 4\%)$ for Reno, Cubic and DCTCP, respectively. *iii)* DCTCP improves the FCT over Reno and Cubic and HSCC further improves the performance of DCTCP.

A Scenario with Background Traffic: to put HSCC under true stress, we run the same One-to-All scenario but we introduce an all-to-all long-lived background traffic during the experiment. We report similar metrics as in the aforementioned case. Figures 12a, 12b, 12c and 12d show the average,



(a) Small: Average FCT (b) Small Flows: Median FCT

(c) Small Flows: Max FCT (d) Small: Missed deadlines

Fig. 12. One-to-All WebSearch with background traffic.

median and max FCT, and missed deadlines for small flows. We observe the following in this case: HSCC improves further the average, median and maximum FCT of small flows regardless of the TCP congestion controller in use. As shown in the results, compared to Reno, Cubic and DCTCP, HSCC reduces the FCT of small flows by $\approx (66\%, 63\%, 26\%)$ on the average, $\approx (52\%, 54\%, 51\%)$ on the median and $\approx (75\%, 75\%, -)$ on the maximum, respectively. HSCC has managed to decrease the maximum FCT and consequently the number of missed deadlines by $\approx (75\%, 75\%)$ for Reno, Cubic, respectively. However, DCTCP sees a slight increase in the max FCT and missed deadlines, which might be attributed to its reaction to excessive ECN marks caused by background flows.

In summary the micro-benchmark and traffic generator experimental results show the performance gains (esp. for time-sensitive applications) obtained by adopting HSCC system. In particular, they show that: 1) It reduces the mean and variance of the FCT of small flows and significantly reduce by 1 to 2 orders-of-magnitude the FCT for the tail end; 2) It can improve the performance further in the presence of bandwidth-hungry long-lived flows; 3) It efficiently handles short-lived traffic, even in low and high frequency incast events; 4) It achieves its goals without requiring modifications to the network stack of the guest VMs.

VI. Related Work

Much work has been devoted to addressing congestion problems in data centers and in particular incast congestion [26], [32], [33], [34]. Recent works [22], [23], [24], [26], [32], [35] analyzed the nature of incast events in data centers and shown that incast leads to throughput collapse and longer FCT. They show in particular that throughput collapse and increased FCT are to be attributed to the data center ill-suited timeout mechanism and use of large initial congestion windows in TCP's congestion control [15], [36], [37], [38].

Towards solving the incast problem, one of the first works [39] proposed changing the application layer by limiting the number of concurrent requesters, increasing the request sizes, throttling data transfers and/or using a global scheduler. Another work [8] suggested modifying the TCP protocol in data centers by reducing the value of the minRTO value from 200ms to microseconds scale. Then DCTCP [4] and ICTCP [40] were proposed as new TCP designs tailored for data centers. DCTCP modifies TCP congestion window adjustment function to maintain a high bandwidth utilization and sets RED's marking parameters to achieve a short queuing delays. ICTCP modifies TCP receiver[4] to handle incast traffic by adjusting the TCP receiver window proactively, before packets are dropped via throughput estimation. These solutions require changes to the TCP logic at the end hosts, can not react fast enough with the dynamic traffic of data centers and impose a limit on the number of senders.

Similar to DCTCP, DCQCN [41] was proposed as an end-to-end congestion control scheme implemented in custom NICs designed for RDMA over Converged Ethernet (RoCE). It achieves adaptive rate control at the link-layer relying on Priority-based Flow Control (PFC) and RED-ECN marking to throttle large flows. DCQCN, not only relies on PFC which adds to network overhead, it introduces the extra overhead of the explicit ECN Notification Packets (CNPs) between the endpoints. TIMELY [6] is another congestion control mechanism for data centers which tracks fine-grained sub-microsecond updates in RTT as network congestion indication. However, its fine-grained tracking increases CPU load on the end hosts and it is sensitive to delay variations on the backward path.

VII. Conclusion

In this paper, we show empirically that the low bandwidth-delay product of data centers results in excessive timeouts. We find that the short TCP loss cycle is one of the major reasons for this. We show analytically that short cycles can greatly degrade TCP performance when the losses at the end of the cycle are only recoverable via timeout. To enhance the performance of short TCP flows, we propose to stretch the period of TCP cycles in data centers and design HSCC, an efficient control theoretic hysteresis switching mechanism. We implemented HSCC as a hardware prototype and tested its performance thoroughly via simulation and experiments in a small testbed data center. The experimental results demonstrated that HSCC improves the FCT of most TCP small flows, which are known to account for the large part of flows generated by data center workloads, without impacting the progress of long-lived flows. Last but not least, HSCC achieves such feat without the need for TCP modifications in the guest VMs.

References

[1] C. Casetti, M. Gerla, and S. Mascolo, "TCP Westwood: End-to-end congestion control for wired/wireless networks," *Wireless Netw.*, vol. 8, pp. 467–479, Sep. 2002.

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

[4] ICTCP [40] implementation as an NDIS driver is only applicable to Windows OS and would not apply to other OSes such as Linux in which case the implementation requires changes to the TCP logic in the Linux kernel.

[3] D. X. Wei, C. Jin, S. H. Low, and S. Hegde, "Fast TCP: Motivation, architecture, algorithms, performance," *IEEE/ACM Trans. Netw.*, vol. 14, no. 6, pp. 1246–1259, Dec. 2006.

[4] M. Alizadeh et al., "Data center TCP (DCTCP)," *ACM SIGCOMM CCR*, vol. 40, p. 63, Aug. 2010.

[5] M. Alizadeh, A. Javanmard, and B. Prabhakar, "Analysis of DCTCP: Stability, convergence, and fairness," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 39, no. 1, pp. 73–84, 2011.

[6] R. Mittal et al., "TIMELY: RTT-based congestion control for the datacenter," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 45, no. 4, pp. 537–550, 2015.

[7] M. Mellia and H. Zhang, "TCP model for short lived flows," *IEEE Commun. Lett.*, vol. 6, no. 2, pp. 85–87, Feb. 2002.

[8] V. Vasudevan et al., "Safe and effective fine-grained TCP retransmissions for datacenter communication," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 39, p. 303, Aug. 2009.

[9] M. Alizadeh, S. Yang, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker, "Deconstructing datacenter packet transport," in *Proc. 11th ACM Workshop Hot Topics Netw.*, 2012, pp. 133–138.

[10] W. Bai, L. Chen, K. Chen, D. Han, C. Tian, and H. Wang, "Information-agnostic flow scheduling for commodity data centers," in *Proc. USENIX NSDI*, 2015, pp. 455–468.

[11] A. Shieh, S. Kandula, A. Greenberg, C. Kim, and B. Saha, "Sharing the data center network," in *Proc. USENIX NSDI*, 2011, p. 23.

[12] E. Zahavi, A. Shpiner, O. Rottenstreich, A. Kolodny, and I. Keslassy, "Links as a service (LaaS) Guaranteed tenant isolation in the shared cloud," in *Proc. Symp. Archit. Netw. Commun. Syst.*, Mar. 2016, pp. 87–98.

[13] B. Stephens, A. L. Cox, A. Singla, J. Carter, C. Dixon, and W. Felter, "Practical DCB for improved data center networks," in *Proc. IEEE INFOCOM Conf. Comput. Commun.*, Apr. 2014, pp. 1824–1832.

[14] W. Bai, L. Chen, K. Chen, and H. Wu, "Enabling ECN in multi-service multi-queue data centers," in *Proc. USENIX NSDI*, 2016, pp. 537–549.

[15] A. M. Abdelmoniem and B. Bensaou, "Hysteresis-based active queue management for TCP traffic in data centers," in *Proc. IEEE INFOCOM Conf. Comput. Commun.*, Apr. 2019, pp. 1621–1629.

[16] M. Mathis, J. Semke, J. Mahdavi, and T. Ott, "The macroscopic behavior of the TCP congestion avoidance algorithm," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 27, no. 3, pp. 67–82, Jul. 1997.

[17] V. Paxson, M. Allman, J. Chu, and M. Sargent. (2011). *Computing TCP's Retransmission Timer*. [Online]. Available: https://tools.ietf.org/html/rfc6298

[18] A. Greenberg et al., "VL2: A scalable and flexible data center network," in *Proc. ACM SIGCOMM Conf. Data Commun.*, Aug. 2009, pp. 51–62.

[19] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken, "The nature of data center traffic: Measurements & analysis," in *Proc. 9th ACM SIGCOMM Conf. Internet Meas.*, Nov. 2009, pp. 202–208.

[20] T. Benson, A. Anand, A. Akella, and M. Zhang, "Understanding data center traffic characteristics," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 40, no. 1, pp. 92–99, 2010.

[21] M. H. J. Keniston and P. S. Panchamukhi. *Kernel Probes (KPROBE)*. Accessed: Nov. 1, 2021. [Online]. Available: https://www.kernel.org/doc/Documentation/kprobes.txt

[22] A. M. Abdelmoniem and B. Bensaou, "T-RACKs: A faster recovery mechanism for TCP in data center networks," *IEEE/ACM Trans. Netw.*, vol. 29, no. 3, pp. 1074–1087, Jun. 2021.

[23] Y. Chen, R. Griffith, J. Liu, R. H. Katz, and A. D. Joseph, "Understanding TCP incast throughput collapse in datacenter networks," in *Proc. 1st ACM Workshop Res. Enterprise Netw.*, Aug. 2009, pp. 73–82.

[24] J. Zhang, F. Ren, and C. Lin, "Modeling and understanding TCP incast in data center networks," in *Proc. IEEE INFOCOM*, Apr. 2011, pp. 1377–1385.

[25] P. Cheng, F. Ren, R. Shu, and C. Lin, "Catch the whole lot in an action: Rapid precise packet loss notification in data center," in *Proc. USENIX NSDI*, 2014, pp. 17–28.

[26] W. Chen, F. Ren, J. Xie, C. Lin, K. Yin, and F. Baker, "Comprehensive understanding of TCP incast problem," in *Proc. IEEE Conf. Comput. Commun. (INFOCOM)*, Apr. 2015, pp. 1688–1696.

[27] D. Borman, R. Braden, and V. Jackbson. (2014). *TCP Extensions for High Performance*. [Online]. Available: https://www.ietf.org/rfc/rfc7323.txt

[28] V. Misra, W.-B. Gong, and D. Towsley, "Fluid-based analysis of a network of AQM routers supporting TCP flows with an application to RED," in *Proc. Conf. Appl., Technol., Architectures, Protocols Comput. Commun.*, Aug. 2000, pp. 151–160.

[29] C. V. Hollot, V. Misra, D. Towsley, and W. Gong, "Analysis and design of controllers for AQM routers supporting TCP flows," *IEEE Trans. Autom. Control*, vol. 47, no. 6, pp. 945–959, Jun. 2002.

[30] H. K. Khalil, *Nonlinear Systems*. Upper Saddle River, NJ, USA: Prentice-Hall, 2002.

[31] Iperf. *The TCP/UDP Bandwidth Measurement Tool*. Accessed: Nov. 1, 2021. [Online]. Available: https://iperf.fr/

[32] A. M. Abdelmoniem and B. Bensaou, "Efficient switch-assisted congestion control for data centers: An implementation and evaluation," in *Proc. IEEE IPCCC*, Dec. 2015, p. 96.

[33] A. M. Abdelmoniem and B. Bensaou, "Enforcing transport-agnostic congestion control in SDN-based data centers," in *Proc. IEEE 42nd Conf. Local Comput. Netw. (LCN)*, Oct. 2017, pp. 128–136.

[34] A. M. Abdelmoniem, B. Bensaou, and A. J. Abu, "HyGenICC: hypervisor-based generic IP congestion control for virtualized data centers," in *Proc. IEEE Int. Conf. Commun. (ICC)*, May 2016, pp. 1–6.

[35] A. M. Abdelmoniem and B. Bensaou, "Curbing timeouts for TCP-incast in data centers via a cross-layer faster recovery mechanism," in *Proc. IEEE INFOCOM Conf. Comput. Commun.*, Apr. 2018, pp. 675–683.

[36] A. M. Abdelmoniem, B. Bensaou, and V. Barsoum, "IncastGuard: An efficient TCP-incast congestion effects mitigation scheme for data center network," in *Proc. IEEE GlobeCom*, Jan. 2018, pp. 1–6.

[37] A. M. Abdelmoniem and B. Bensaou, "Incast-aware switch-assisted TCP congestion control for data centers," in *Proc. IEEE Global Commun. Conf. (GLOBECOM)*, Dec. 2014, pp. 1–6.

[38] A. J. Abu, B. Bensaou, and A. M. Abdelmoniem, "Inferring and controlling congestion in CCN via the pending interest table occupancy," in *Proc. IEEE Local Comput. Netw. (LCN)*, Nov. 2016, pp. 433–441.

[39] E. Krevat et al., "On application-level approaches to avoiding TCP throughput collapse in cluster-based storage systems," in *Proc. 2nd Int. Workshop Petascale Data Storage, Held Conjunct Supercomputing*, Nov. 2007, pp. 1–4.

[40] H. Wu, Z. Feng, C. Guo, and Y. Zhang, "ICTCP: Incast congestion control for TCP in data-center networks," *IEEE/ACM Trans. Netw.*, vol. 21, no. 2, pp. 345–358, Apr. 2013.

[41] Y. Zhu et al., "Congestion control for large-scale RDMA deployments," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 45, no. 4, pp. 523–536, Aug. 2015.

Ahmed M. Abdelmoniem (Member, IEEE) received the Ph.D. degree in computer science and engineering from the Hong Kong University of Science and Technology, Hong Kong, in 2017. He was a Research Scientist with KAUST, Saudi Arabia, and a Senior Researcher with Huawei's Future Network Laboratory, Hong Kong. He is an Assistant Professor with the Queen Mary University of London, U.K. and Assuit University, Egypt. His work appears in top-tier conferences and journals. His research interests include distributed systems and networks and machine learning.

Brahim Bensaou (Senior Member, IEEE) received the Ph.D. degree in computer science from University Paris VI in 1993. He was a Research Assistant with France Telecom Research Laboratories, a Research Associate with HKUST, and a Senior Staff with the National Research and Development Centre for Wireless Communications in Singapore. He is a Faculty Member with the CSE Department, HKUST. He published more than 130 papers in prominent conferences and journals, received numerous research grants, and supervised more than 20 PG students. His research interests include centered around internet, wireless and mobile communications, and their performance (e.g., congestion control, energy efficiency, and performance evaluation).