Hysteresis-based Active Queue Management for TCP Traffic in Data Centers

Ahmed M. Abdelmoniem
CSE Dept., HKUST, Hong Kong
CS Dept., FCI, Assiut University, Egypt
amas@cse.ust.hk

Brahim Bensaou
Dept. of Computer Science and Engineering
HKUST, Clear Water Bay, Hong Kong
brahim@cse.ust.hk

Abstract*—Much of the incremental improvement to TCP over the past three decades had the ultimate goal of making it more effective in using the long-fat pipes of the global Internet. This resulted in a rigid set of mechanisms in the protocol that put TCP at a disadvantage in small-delay environments such as data centers. In particular, in the presence of the shallow buffers of commodity switches and the short round trip times in data centers, the continued use of a large TCP initial congestion window and a huge minimum retransmission timeout (both inherited from the Internet-centric design) results in a very short TCP loss cycle that affects particularly the flow completion times of short-lived incast flows. In this paper, we first investigate empirically the TCP loss cycle and discuss its impact on packet losses, recovery and delay; then we propose a switch-based congestion controller with hysteresis (HSCC) that aims to stretch the TCP loss cycle without modifying TCP itself. To protect incast flows from severe congestion, HSCC is designed to transparently induce the TCP source to alternate between its native TCP congestion control algorithm and a slower more conservative constant bit rate flow control mode that is activated when congestion is imminent. We show the stability of HSCC via analytical modelling, and demonstrate its effectiveness via simulation and implementation in a small testbed.

***Index Terms*—Congestion Control, Hysteresis, AQM, TCP**

I. Introduction

TCP is by far the most predominant transport protocol in use in today's data centers. It was gradually fine-tuned with additional mechanisms that were not part of its initial incarnation, mostly in response to the evolution of the Internet scale (distance and bandwidth). As a direct consequence, most default TCP implementations found in the most popular operating systems today are geared to be efficient in the high-bandwidth long-delay Internet environment. In particular, the congestion control algorithm has seen dramatic changes over the years and numerous TCP congestion controllers have seen the light, mostly to meet the requirements of new operating environments in the Internet (e.g., Cubic TCP [1], Fast TCP [2], and so on).

With such protocols, it has been observed that small short-lived flows experience unduely long flow completion times (FCT) in data centers, with a negative impact on the user experienced performance. As a consequence, an increased attention has been paid to addressing this problem in the past few years. For instance, DCTCP [3, 4], and TIMELY [5], propose new congestion control mechanisms designed specifically to work well in data center networks. In contrast, other studies, simply identified the source of performance degradation and proposed to tune existing congestion control parameters to match the scale of data center networks (e.g., reduce the initial congestion window to cope with the small switch buffers [6] or scale down the minimum retransmission timeout to match the typically small RTT of data center [7]). All these approaches have been shown to yield some performance improvements, and some are already in use in production data centers, however, these solutions can only apply to privately owned homogeneous data centers where the operator controls both ends of the internal TCP connections and can replace the transport protocol with a new one in all the virutal machines. Switch assisted congestion control has also been investigated as a means to improving the FCT of short-lived flows. For example pFabric [8], and PIAS [9] leverage priority queuing in the switches to segregate and serve short-lived flows with a high priority. These mechanisms also apply exclusively to privately owned data centers as they require modification of the end-system in addition to the switch (e.g., PIAS [9] relies on DCTCP).

In virtualized multi-tenant, data centers, the common physical infrastructure is shared by multiple tenants that run their applications on virtual machines (VMs). The tenants can implement and deploy their preferred version of operating system and thus of TCP, or even opt for using UDP. Also, the end user can tweak TCP parameters in the guest VM to meet the application needs. As a result, the approaches described above cannot apply as they work only in homogeneous data centers. To tackle this problem, several approaches were proposed in the literature: in the most straightforward, the public data center operator statically divides the network bandwidth among its tenants, giving each of them a fixed allocation with guaranteed bounds on delays [10, 11]. This technique, though effective, would ignore statistical multiplexing resulting in a small inefficient flow admissible region, in view of the burstiness of the traffic [12]. The second approach suggests to modify all the switches in the data center to enforce a small buffer occupancy at each switch. This can be achieved by using a form of weighted fair queuing and/or by applying various marking thresholds within the same queue similar to DiffServ [13, 14]. Typically, each source algorithm requires a certain

This work was done while Ahmed was at CSE Dept., HKUST, HK and was supported in part under grant: HKPFS PF12-16707.

weight/threshold to fully utilize the bandwidth. Hence, such schemes are not scalable to deal with the large numbers of flows that share public data center networks. In addition, they may lead to the starvation of some traffic classes in the absence of flow admission control and are hard to deploy due to the increasing number of congestion control algorithms employed by the tenants. The last approach is covered in recent works in [15, 16] and aims to enable virtualized congestion control in the hypervisor or by enforcing it via the virtual switch (vswitch) transparently to the tenant VM. This last approach requires fully-fledged TCP state tracking and implements full TCP finite-state machines in the hypervisor which eventually can overload and slow down the hypervisor considerably.

To enable true deployment potential in such heterogeneous TCP environment without changing TCP, in this paper, we adopt a switch-based approach that interacts with the end-systems only via standard universally adopted TCP mechanisms to convey congestion signals to the sources, which makes it independent of the actual congestion control algorithm implemented at the source. In normal mode when there is no congestion, the sources compete via the TCP congestion controller, whereas when congestion is imminent, the sources become simple flow controllers. The alternation between these two modes is controlled by the switch and is triggered in response to congestion. This approach enables public data center operators to innovate in the switch without paying attention to the TCP variations running in the VMs.

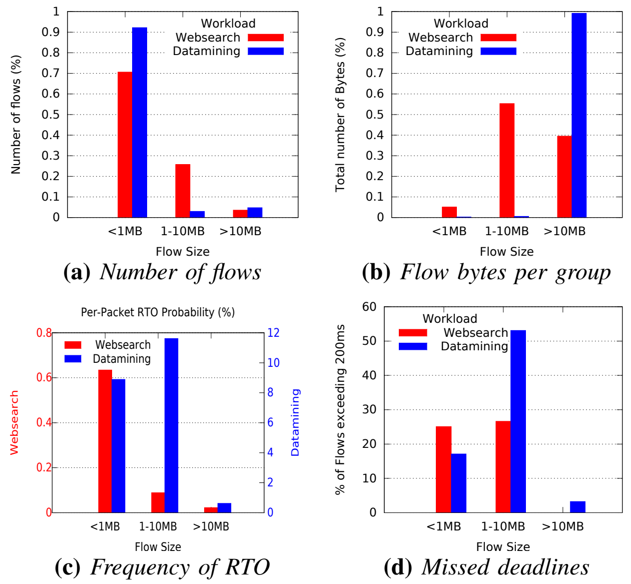
In the remainder of the paper, we investigate and discuss the role played by RTO in the degradation of the FCT performance of short-lived flows in Section II. We dissect the problem and propose our solution in Section III. In Section IV we model our control to show its stability, and discuss some of its design and practical aspects. In Section VI, we provide some simulation results, then, in Section VII, we discuss some implementation details and show experimental results from a real small-scale testbed. We discuss some important related work in Section VIII. And, finally we conclude the paper in Section IX.

II. Motivation

A. Impact of RTO on The FCT

In data centers, partition/aggregate applications that generate short-lived flows are challenged by the presence of small buffers, large initial sending windows, inadequate minRTO and/or slow-start exponential increase. This combination of hardware and TCP configuration frequently leads to timeout events for such applications. In particular, when the number of flows they generate is large and roughly synchronized, incast-TCP synchronized losses occur. As the loss probability increases linearly with the number of flows [17], the flow synchronization and the excessive losses lead to throughput-collapse for small-flows in data centers.

To illustrate this, consider a simplified fluid-flow model with N flows sharing equally a link of capacity C. Let B be the flow size in bits and n be the number of RTTs it takes to complete the transfer of one flow. The optimal throughput



(a) *Number of flows* **(b)** *Flow bytes per group* **(c)** *Frequency of RTO* **(d)** *Missed deadlines*

Figure 1: *RTO frequency and its impact on the FCT.*

ρ^* can be simply expressed as the fraction of the flow size to its average transfer time: $\rho* = \frac{B}{n\tau + \frac{BN}{C}}$. That is, it takes BN/C to transmit the B bits plus an additional queueing and propagation delay of τ seconds for each of the n RTTs. In practice, when TCP incast congestion involving N flows results in throughput-collapse, the flow experiences one or more timeouts and recovers after waiting for RTO. Then, the actual throughput writes: $\rho = \frac{B}{mRTO + n'\tau' + \frac{BN}{C}}$, Typically $n' \geq n$ and $\tau' \geq \tau$. In addition, in data centers, the typical RTT is around $100\mu s$, while existing TCP implementations impose a minimum RTO of about 100 to 200ms. As a consequence, large sized flows yield values of n' such that $n'\tau$ is similar or greater than RTO. In contrast small short-lived flows only last for a few RTTs, therefore $RTO \gg n'\tau$. And so, when a small flow experiences a loss that cannot be recovered by 3-duplicate ACKs (called in the sequel a non-recoverable loss or NRL), then it has a high chance of missing for example its service level agreement deadline (of say $\approx$ 100ms). To improve the performance of small flows without altering TCP, curbing NRLs as much as possible for such flows is the answer. This can be achieved by adapting the source rate of all ongoing flows when congestion is imminent to make room in the switch buffer for small flows.

B. Measurement-Based study

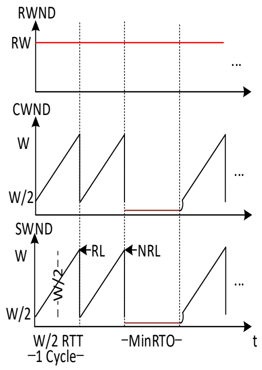
To reinforce this simple analysis, we studied empirically the frequency of timeouts in a small-scale testbed equipped with data center grade servers and top-of-rack switches. To make the results meaningful, we reproduced the workloads found in public and private data centers via a custom built TCP traffic generator, based on flow sizes and inter-arrival time distributions drawn from various realistic workload studies (e.g., Websearch [3] and Datamining [18] and others [19, 20]. To track the nature of packet losses, we custom built a Linux kernel module to collect live TCP socket-level events and statistics (e.g., timeouts, retransmissions, sequence numbers, and so on). We generated a total of 7000 flows and categorized them into small flows (with a size $\leq 1MB$), medium flows(with a size of 1 to $10MB$) and large one (with sizes exceeding $10MB$). We use TCP New-Reno without ECN function.

Similar to past works, Fig. 1a and Fig. 1b show that, in Websearch and Datamining workloads, most flows are small. In addition, in web search data bytes are distributed almost uniformly over the three categories, whereas in Datamining, most of the bytes are produced by large flows. Fig. 1c shows the per-packet RTO probability for each type and suggests that RTO is highly likely for small and medium flow types in both workloads. Noticeably, the per-packet RTO probability (i.e., to recover from NRL) for small flows is almost $\approx 0.6\%$ and $\approx 9\%$ in Websearch and Datamining workloads, respectively. The RTO probability is non-negligible as it correlates with the number of flows missing their deadlines (e.g., their FCT exceeds the ideal FCT by an extra 200ms) which, according to Fig. 1d, is $(\approx 26\%, \approx 18\%)$ in Websearch and Datamining, respectively.

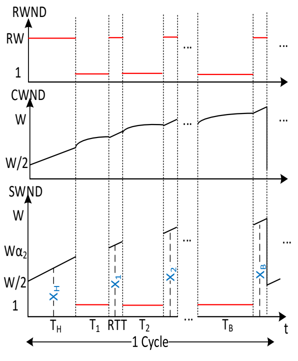
As an example, assuming on average a small flow size is B=500KB and on average 36 flows [3] share the bottleneck link capacity of C=1Gbps equally, then, such flow should finish its transmission in $\approx$ 17 RTTs with an FCT of about $150ms$. According to Fig. 1, on average a flow would experience 1/2 RTO in Websearch (respect. 2 RTO in Datamining) delay. This translates into adding more than $100ms$ (respect. $400ms$) to the ideal FCT for Websearch (respect. Datamining). These results show the effect of RTOs on small flows that usually have just a few segments to send.

III. Anatomy of the Problem and Solution

In this section, we explore the problem of FCT bloating due to RTOs and its impact on the throughput.



(a) *TCP Droptail AQM*



(b) *TCP Hystersis-based AQM*

Figure 2: *(a) The period and the number of MSS sent within each TCP loss cycle. (b) Same TCP interleaved with intra-cycle slow CBR modes which stretches the loss cycle.*

A TCP flow sending rate is controlled by the sending window size $swnd$ which is drawn as the minimum of two other windows: the congestion window, $cwnd$, designed to estimate the steady state bottleneck-link fair-share of the flow, and the flow control receiver window, $rwnd$, carried by returning Acknowledgement packets, indicating the extra number of bytes that the receiver can accept at this time. In TCP congestion avoidance (CA), $cwnd$ evolves in a periodic sawtooth shape as a result of the well known AIMD algorithm which ensures that $cwnd$ attains all the values between the maximum (w) and its minimum value $(\frac{w}{2})$ [17]. The value w evolves downward when congestion increases and upward when congestion recedes. In modern data center, because the RTT is small, $rwnd \gg cwnd$, in other words the flow control window has no effect and the source sending rate is fully determined by the congestion window. In addition, the bandwidth delay product is small, therefore each TCP source's bandwidth fair share is also small. The throughput of TCP being inversely proportional to the square root of the loss event probability [17], such small share implies frequent losses. For incast traffic upon arrival, the short loss cycle implies that some flows will not be able to build a large enough flight size to recover via 3-duplicate ACKs. This leads to repeated NRLs and waiting for minRTO. Fig. 2a illustrates this short loss cycle of TCP. This problem can take place with all window-based TCP congestion control mechanisms that rely on loss/congestion signals to adjust their sending rates (e.g., RENO, CUBIC, DCTCP).

A. A Control Theoretic Solution

We propose a simple solution that aims to avoid non-recoverable losses by delaying losses outright (or stretching the loss cycle). Our control mechanism enables the sources to alternate between two operation modes: one governed by the standard AIMD, in which the source sends data according to its congestion window $cwnd$ as before, and one conservative constant bitrate mode (CBR) where each source is only allowed to send a small constant number of segments per RTT. Alternation between the two modes happens in response to signals sent by the switch, based on the congestion level observed in the switch buffer. When the queue in the switch buffer builds up, the switch triggers the CBR mode. This can be achieved by relying on the flows control windows $rwnd$ to enforce the CBR rate (for example the controller can simply rewrite the value of $rwnd$ in the ACK headers to 1 MSS for all flows). When the queue recedes, the senders resumes the use of $cwnd$ allowing them to recover the previous sending rates.

An implicit consequence of this scheme is that short-lived incast traffic is discriminated positively when it is most likely to experience non-recoverable losses, immediately after the connection is set up. That is, when many synchronized flows surge, the buffer content builds up fast, and our scheme switches all ongoing flows to CBR mode. These flows react, typically $\frac{1}{2}$RTT later, by reducing their sending rate to the minimal CBR rate of 1 MSS while the incast traffic flows are still within their three-way handshake (or sending their first few packets). By dynamically modifying $rwnd$, the switch implicitly inhibits $cwnd$ without modifying TCP in the end systems, and thus controls when the TCP source sends according to its fairly acquired $cwnd$ value and when it sends according to the conservative rate set by the switch in $rwnd$. Fig. 2(b) shows $cwnd$, $rwnd$ and $swnd$ of TCP with periods where $cwnd$ is inactive.

IV. HSCC System Modeling

HSCC control loop is depicted in Fig. 3a, the system consists of five components namely three data transfer modes, the queue and a hysteresis controller that switches dynamically between the three data sources. These include a TCP Additive increase source with increase rate of 1 MSS/RTT, a CBR source sending at a constant rate of 1 MSS/RTT and an artificial CBR+ source that combines the previous CBR source and a Multiplicative Decrease on the congestion window. Fig. 3b shows the switching law used by HSCC. HSCC is a Counter-Clockwise (CC) hysteresis where the switching happens first when the high threshold α_2 is crossed then the state remains the same until the lower threshold α_1 is crossed. Fig. 3b shows the sequence of switching as follows: *i)* while using TCP source sending at rate λ_1, when the high threshold α_2 is hit, the controller switches to the lower rate CBR source with rate λ_0; *ii)* the system keeps operating in this CBR mode with rate λ_0 until the lower threshold α_1 is hit. At this point, the controller switches back to the TCP source with sending rate λ_2. Notice that, as $cwnd$ is only inhibited during CBR mode, the TCP source will continue increasing $cwnd$ with each ACK during the CBR mode, leading to the new rate λ_2. *iii)* The system continues like this until the queue exceeds the buffer space (i.e., M) leading to the loss of packets. In such case, the system switches to a third source CBR+ where it applies multiplicative decrease to $cwnd$ and switches to send at rate λ_0 until it crosses again the low threshold. *iv)* otherwise, the system stays in the current state. Fig. 3c shows the system transition diagram.

A. HSCC Design and Implementation

Fig. 4 shows the system components which are: 1) the switch, which triggers the switching between the different operating modes and rewrites the receive window field $rwnd$ to 1 MSS on the ACKs for all flows for ports operating CBR mode. Hence, the switch does not require any per-flow state. 2) the end-host helper module that resides in the hypervisor and whose role is to avoid misalignment in the $rwnd$ field when TCP window scaling factor is activated by the end system.

HSCC Operational Aspects: The end-host helper module hashes flows into a hash-table with the flow's 4-tuples (source IP, dest. IP, source port and dest. port). The hash is used as the key and the corresponding window scale factor as the value. Flow entries are cleared from the table when a connection is closed (i.e., FIN is sent out). To ensure the scale factor is taken into account, the end-host module writes the scale factor for all outgoing ACK packets in the 4-bit reserved field of TCP headers (alternatively, we could use 4-bits of the receive window field and use the remaining 12 bits for window values). The reserved bits are cleared by the helper module otherwise the TCP checksum is invalid and the packet is dropped. This approach avoids the need for recalculating a new checksum at the end-host and the switch.

As shown in Fig. 4, the end-host 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 placement does not touch any network stack implementation of host nor guest operating system, making it ready for deployment in production data centers. The end-host module tracks the scaling factor used by local communicating end-points and explicitly append this information only to outgoing ACKs of the corresponding flow. The switch module on the other hand monitors the output port queues and starts rewriting the incoming ACKs headers according to the HSCC switch dynamics. To avoid problems at the receiver, the switch uses the appended scale factor to rescale the window value so that it can be interpreted correctly by the ACK receiving end-point in the VM.

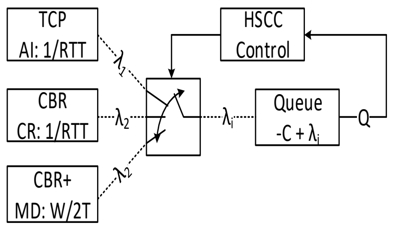
Receive Window Scaling: HSCC needs a hypervisor module as it relies on a scale factor to rescale the modified window written into TCP header of incoming ACKs. TCP specification [21] states that the three-byte scale option may be sent in all packets or only in a SYN segment by each TCP end-point to inform its peer of its own scaling factor. If the former approach is adopted, then the hypervisor module is not necessary as the switch can readily obtain this value from the packet itself, however, TCP implementations in most operating systems including Linux adopt the latter approach to cut overhead. Also, in practice, window scaling may be unnecessary for networks with Bandwidth-Delay (BD) product of 12.5KB (i.e., C=1 Gbps and RTT$\approx$ 100μs). However, with the adoption of high speed links of 10 Gbps (i.e., BD=125KB), 40 Gbps (i.e., BD=500KB) and 100 Gbps (i.e., BD=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.

The probability of having such small number of active flows in data centers is extremely small [3], but still possible. As such in HSCC we opt to still handle window scaling via the end-host module instead of doing it in the switch. The shim-layer extracts and stores from outgoing SYN and SYN-ACK packets the advertised scaling factor for each established TCP flow and encodes the scale factor using the 4 reserved bits in the TCP header. The switches on the path use this value to scale the new receive window properly if needed while the destination end-host module clears it out before the packet is forwarded to upper layers or the guest VM.

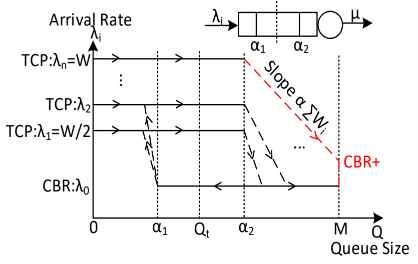
Need for Symmetric Routing: HSCC by design requires the ACKs to return on the same backward path the data flows through. This requirement is easily met given the common deployment of ECMP routing in data centers [18–20].

V. Stability Analysis

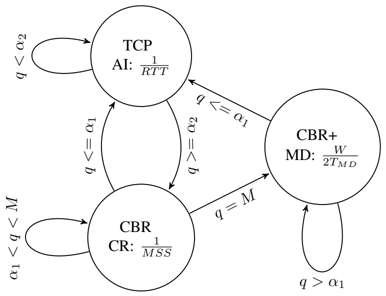
We can easily show that the combination of TCP and HSCC switch forms a non-linear system that is stable. To this end we invoke the fluid flow modeling approach used in [22]



(a) *A schematic diagram of the system*



(b) *HSCC hysteric control low*



(c) *State transition diagram*

Figure 3: *(a) HSCC system components and the feedback loop shows the hysteresis controller managing the switching between TCP, CBR and CBR+ sources (b) the control law of HSCC obeying a counter-clockwise hysteresis to switch between states based on queue occupancy . (c) Flow chart depicting the possible state transition of HSCC system*

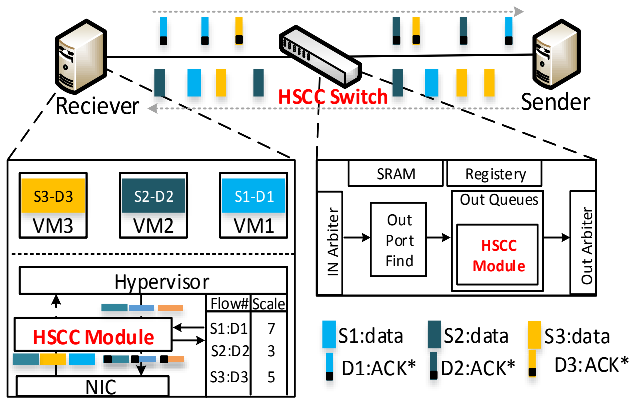


Figure 4: *HSCC System: It consists of an end-host module that attaches scaling value to ACKs and HSCC switch which performs Hysteresis switching between TCP and CBR.*

with standard linearization [23]. Due to space constraints we only give here a sketch of the proof without going into all the details. We assume a finite buffer capacity M, and the RTT differences among competing flows to be negligible, packet size L is constant and sources have an infinite supply of data (i.e., small flows become a temporal low-frequency disturbance/noise imposed on the system, and are absorbed by system dynamics). The RTT for a packet on a bottleneck link of capacity of C can be written as $\tau_i(t) = T_c + T_t + T_p + \frac{q_i(t)}{C}$, where $T_t = \frac{size(L)}{C}$ is the transmission time, T_p is the propagation delay, T_c is the processing delay on the path and $\frac{q_i(t)}{C}$ is the queueing delay seen by flow i. Let $P_l \in [0, 1)$ be the packet drop probability triggering 3-DUPACK recovery and $P_u \in [0, 1)$ be the update probability of $rwnd$ when the hysteresis switch is ON (i.e., CBR mode is active). Under these assumptions the fluid model of TCP-HSCC can be shown to be a non-linear system with two variable $(w(t), q(t))$ and two inputs (P_l, P_u). Its dynamics are governed by the following system of differential equations:

$$\begin{aligned}\frac{dw(t)}{dt} &= \left(\frac{1 - P_u(t-\tau(t))}{\tau(t)} + \frac{P_u(t-\tau(t))}{w(t)\tau(t)}\right)\frac{w(t-\tau(t))}{\tau(t-\tau(t))} \\ &\quad - \frac{w(t)P_l(t-\tau(t)}{2}\frac{w(t-\tau(t))}{\tau(t-\tau(t))}, \\ \frac{dq(t)}{dt} &= (1 - P_u(t-\tau(t))N\frac{w(t)}{\tau(t)} + P_u(t-\tau(t))N\frac{1}{\tau(t)} - C. \end{aligned} \tag{1}$$

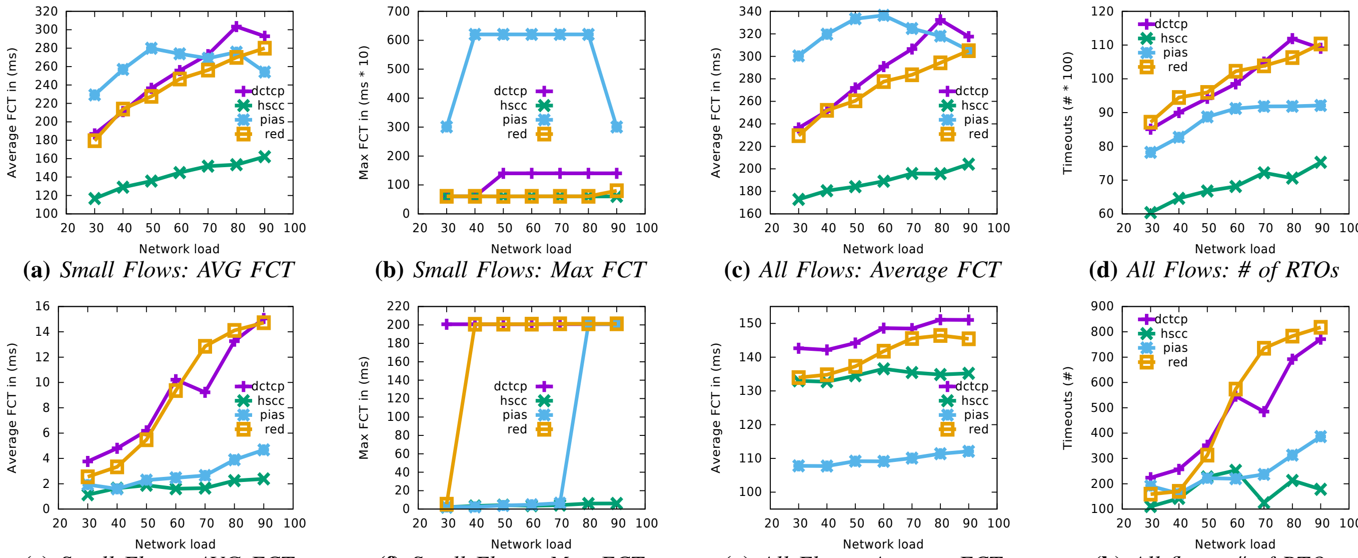
Linearization: By definition of the FIFO queue, we can immediately write, if $\alpha_2 \le M$ then $P_u \ge P_l$. For convenience we write $P_u = \kappa P_l$ with $\kappa \in (0, 1]$. Intuitively, κ is a positive scalar inversely proportional to the number of queue passages over high threshold α_2 before a single buffer overflow takes place. Since after every passage through α_2, when the TCP mode becomes active the window increases by 1 MSS per flow and if N (the number of flows) is given then κ can be calculated. Hence, κ is the number of packets between the high threshold α_2 and the full buffer M divided by the number of flows N (i.e., $\kappa = \frac{(1-\alpha_2)M}{N}$). The operating point is when the system dynamics comes to rest (i.e., at equilibrium point defined by $(w_0, q_0, P_{l0} = P_0, P_{u0} = \kappa P_0)$. Hence, the equilibrium points can be found by solving Eq.(1) for $\frac{dw}{dt} = 0$ and $\frac{dq}{dt} = 0$ (The details of the linearization procedure are similar to [22] and are omitted.)

$$\begin{aligned}\frac{dw(t)}{dt} = 0 \quad &\rightarrow \quad P_0 = \left(\frac{\kappa w_0 \tau_0}{2} - \frac{1}{w_0} + 1\right)^{-1} \\ \frac{dq(t)}{dt} = 0 \quad &\rightarrow \quad W_0 = \frac{C\tau_0/N - P_0}{1 - P_0},\end{aligned} \tag{2}$$

where $\tau_0 = \frac{q_0}{C} + T$. We use the obtained equilibrium points to define the perturbed variables and inputs as $\delta w = w - w_0$, $\delta q = q - q_0$ and $\delta P_u = P_u - P_{u0}$. Then we can construct the linearized system of equations as follows:

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

From here, we can easily show that the TCP-HSCC system is stable by simply showing that the dynamics matrix of 3 is



(a) *Small Flows: AVG FCT* **(b)** *Small Flows: Max FCT* **(c)** *All Flows: Average FCT* **(d)** *All Flows: # of RTOs*

(e) *Small Flows: AVG FCT* **(f)** *Small Flows: Max FCT* **(g)** *All Flows: Average FCT* **(h)** *All flows: # of RTOs*

Figure 5: *Performance metrics of simulation runs in a large 9 Leaf - 4 Spine topology. The traffic generator varies the network load in the range of [30%, 90%]. (a-d) Websearch workload. (e-h) Datamining workload.*

Hurwitz stable [23].

VI. Simulation Analysis

We first conduct a simulation analysis of HSCC performance and then we will examine experimental results from a real implementation in a small cluster.

A. *Realistic Traffic in DC Topology*

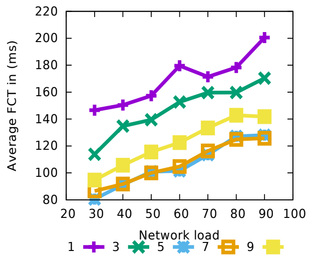
We conduct simulation analysis of HSCC system in a data center-like topology with varying workloads and flow size distributions. We use a spine-leaf topology with 9 leaves and 4 spines using link capacities of 10Gbps for end-hosts with an over-subscription ratio of 5 (the typical ratio in current production data centers is in the range of 3-20+). Each rack hosts 16 servers with a total of 144 servers and per-port queue of 84 packets. We compare the performance of HSCC with various alternative schemes discussed in Section II (e.g., TCP with RED-ECN, DCTCP and PIAS). The performance metrics of interest are the flow completion time or FCT for mice flows (i.e., flows in the range [0-100Kbtyes]), the average FCT for all flows, the number of timeouts experienced and the number of unfinished flows. In the simulations, per-hop link delays are set to 50 μs, TCP is set to the default TCP RTO_{min} of 200 ms and TCP is reset to an initial window of 10 MSS, and a persistent connection is used for successive requests (similar to the Linux implementation). The flow size and inter-arrivals distribution are extracted from two workloads (i.e., Websearch and Datamining). A parameter (λ) is used to simulate various network loads. Buffer sizes on all links are set to be equal to the bandwidth-delay product between end-points within one physical rack. The low threshold α_1 is set to 25% and the high threshold α_2 is set to 50% of the buffer size.

Fig. 5 shows the average and maximum FCT for small flows as well as the average FCT and total timeouts of all flows. The results show the performance of the four schemes. We observe that HSCC can greatly improve the average and tail FCT of small flows. As a result, the average FCT of all flows is improved for two reasons: small flows are larger in number and they can finish quicker leaving network resources for large ones. HSCC helps in reducing the number of timeouts, which improves the average and max FCT. The results suggest that stretching loss cycles can lead to significant performance gains. We note that PIAS performs better in Datamining workload and worse in Websearch workload. We suspect that the fixed so-called demotion threshold of PAIS and larger flow sizes in Websearch lead to starvation of certain flows. We inspected the output traces and found that across the loads [30-90]%, PIAS has 25 unfinished (or starved) flows.

B. *Sensitivity Analysis of HSCC*

We repeat the initial simulation experiment using Websearch workload with the same parameters as previously while varying the values of low threshold $\alpha 1$ and high threshold $\alpha 2$ to assess the sensitivity of HSCC to the setting of these parameters. The results (omitted due to space) show that FCT is not affected at all by the choice of the parameter $\alpha 1$ and $\alpha 2$. Similar results are observed for Datamining. This is not surprising because in all cases the system switches between low rate CBR and TCP but at slightly (sub-microsecond) different times. This means that our scheme is robust for any reasonable values of the thresholds.

We repeat the experiment while varying the values of $rwnd$ used in the rewriting process of the receive window between 1-10 MSS. As Fig. 6 shows, the FCT greatly depends on the choice of this value. The improvement forms a normal distribution with increase starting at 1 MSS up to a peak at 5 MSS followed by a degradation of the improvement. Further study of the optimal value of $rwnd$ is left for future work.


(a) *Small Flows: AVG FCT*

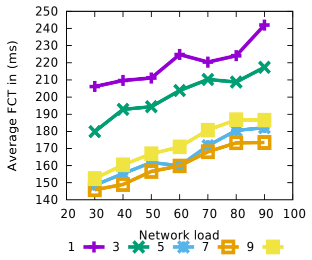

(b) *All Flows: AVG FCT*

Figure 6: *A websearch experiment to show the sensitivity of HSCC to the choice receive window* ***RWND***.

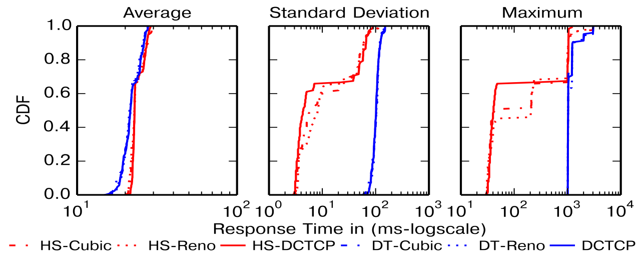
VII. Experimental Analysis

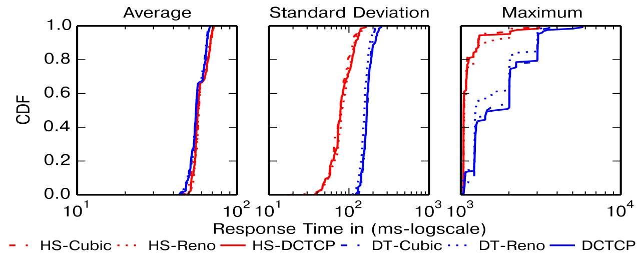
We prototyped the HSCC controller on the NetFPGA platform and used it to conduct a series of experiments.

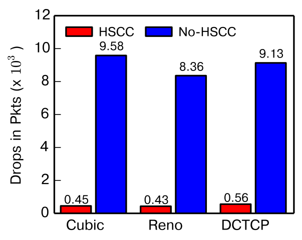
The testbed consists of 14 high performance Dell PowerEdge R320 servers. The machines are equipped with Intel Xenon E5-2430 6-cores CPU, 32 GB of RAM. Each server has a built-in two 1 Gb/s Broadcom NICs in addition to an Intel I350 server-grade 1 Gb/s quad-port NIC resulting into 6 NICs per server. Hence, using the 14×6 NICs, we organized the servers into 4 racks (each rack is typically a subnet) and connected them via 4 non-blocking Top-of-Rack (ToR) switches. We implemented HSCC on a NetFPGA and deployed it as a core switch to interconnect the 4 ToR switches. The 4 racks are divided into (rack 1, 2 and 3) as senders and rack 4 as the receiver. Each port in the same subnet is connected to one of the non-blocking ToR switches through 1 Gb/s link. The servers are installed with Ubuntu Server 14.04 LTS upgraded to kernel version (3.18) which has by default the implementation of DCTCP [24], Cubic and New Reno congestion control mechanisms. Finally, we also loaded and run on all end-hosts the HSCC shim-layer implemented as Linux netfilter kernel module.

A. Micro-Benchmarking Experiments

For experimentation purposes, the machines are installed with the iperf program [25] for creating long-lived traffic (i.e., elephant flows) and the Apache web server and Apache benchmark [26] for creating small web responses (i.e., mice flows). We setup different scenarios to reproduce both "incast" and "incast with background workload" situations. To reduce CPU load, guest VMs are emulated via host processes, each process being bonded to a virtual port created on the Open vSwitch (OvS) [27]. These processes are either an iperf flow or an Apache client/server process bonded to its own virtual port. In this manner, we can emulate traffic originating from any number of VMs and simplify the creation of scenarios with a large number of flows in the network. The micro-benchmark objectives are as follows: *i)* to verify that with the support of HSCC, TCP can support many more connections and maintains high link utilization; *ii)* to verify the effectiveness of HSCC system in reducing incast congestion effect on TCP flows; *iii)* to observe HSCC's ability to improve the FCT of mice when competing for the bottleneck link with elephants.


(a) *Mild traffic case: 126 mice flows retrieving 1.45GB*


(b) *Heavy traffic case: 630 mice flows retrieving 7.25GB*


(c) *Mild incast case*

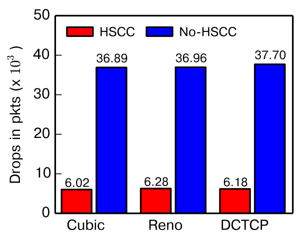
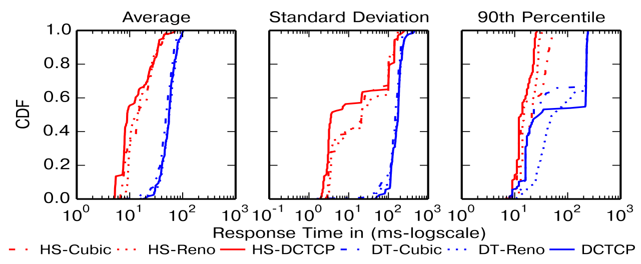

(d) *Heavy incast case*

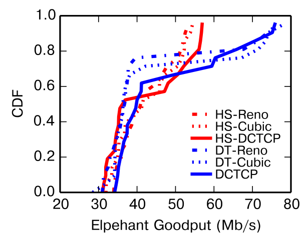
Figure 7: *Experimental results of two Incast mild and heavy traffic scenarios.*

Incast Traffic without Background Workload: First, we run two mild and heavy incast scenarios where a large number of mice flows transfer 11.5KB sized blocks. In both scenarios, 7 servers in rack 4, issue web requests for **"index.html"** of size 11.5KB from another 21 servers in rack 1, 2 and 3. Hence, a total of 126 (21 × 7 − 21) synchronized requests are issued. In the mild scenario, each request is repeated a thousand times consecutively, which is equivalent to an 11.5 MB transfer. The scenario involves the transfer of 1.5GB in total (i.e., $11.5MB \times 126$) within a short period through the bottleneck link. In the heavy load case, a thousand consecutive requests are issued however, each process uses 5 parallel TCP connections instead of one only. This results in 630 flows (i.e., 126×5) at the same time. Statistics of the FCT for mice flows are collected from Apache benchmark.

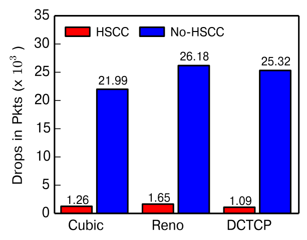
Fig. 9 shows, under both mild and heavy load, HSCC achieves a significantly improved performance. The competing mice flows benefit under HSCC in the mild case by achieving almost the same FCT on average but with an order-of-magnitude smaller standard deviation compared to TCP (Cubic, Reno) with DropTail and DCTCP. In addition, HSCC can improve the tail FCT (max-FCT) by two orders-of-magnitude, suggesting that almost all flows (including tail ones) can meet their deadlines. In the heavy traffic case, it can also achieve noticeable improvements even with 630



(a) *126 mice flows' performance in 1 incast epoch*

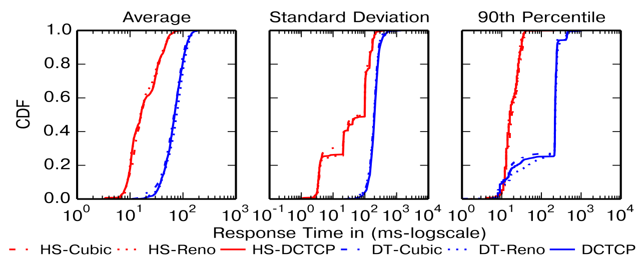


(b) *AVG elephant throughput*

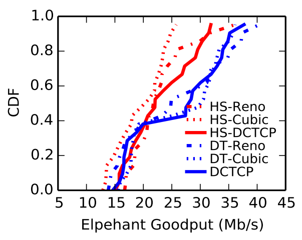


(c) *Total packet drops*

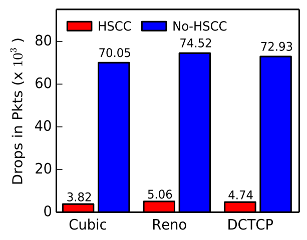
Figure 8: *Performance of HSCC vs (TCP with DropTail or DCTCP): each of the 126 mice flow requests 1.15MB file (= $100 \times 11.5KB$) 1 time while competing with 21 elephants*



(a) *126 mice flows' performance in 9 incast epochs*



(b) *AVG elephant throughput*



(c) *Total packet drops*

Figure 9: *Performance of HSCC vs (TCP with DropTail or DCTCP): each of the 126 mice flow requests 1.15MB file (=$100 \times 11.5KB$) 9 times while competing with 21 elephants*

competing flows, each with 1 MSS (1460 bytes) of window worth, totalling 920KB $\approx 3.2\%$ times more than the size of the bottleneck link of 287KB (the switch buffer size plus the bandwidth delay product). Finally, HSCC can efficiently react to incast and proactively throttles the flows to avoid packet drops, Fig. 9 shows that, it can significantly decrease the drop rate during incast events by $\approx 96\%$ in the mild load compared to only $\approx 86\%$ in the heavy load scenarios.

Incast Traffic with Background Workload: We need to characterize HSCC performance when it is subjected to background long-lived flows and its effect on elephant flows' performance. To this end, incast flows are set to compete with elephant flows for the same outgoing queue. For this, 21 iperf [25] long-lived flows are set to send towards rack 4 continuously for 20 secs. In this case, the incast flows must compete 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 1.15MB file partitioned into 100 11.5KB chunks totalling $\approx 145MB$) after elephants have reached their steady state (i.e., at the 10^{th}sec). Fig. 8a shows that, HSCC achieves FCT improvements for mice flows while nearly not affecting elephant flows performance. Mice flows benefit with HSCC by improving the FCT on average and with one order-of-magnitude reduction in FCT standard deviation compared to TCP (Cubic, Reno) with DropTail and DCTCP. Also, HSCC reduces the tail FCT by two order-of-magnitude, making it nearly close to the average. The improvement means that mice flows finish quickly within their deadlines. Fig. 8b shows that elephant flows are almost not affected by HSCC's intervention due to the throttling of their rates during the short incast periods. In Fig. 8c, packet drops under HSCC are shown to be reduced because of its effective rate control during incast and hence mice flows avoid long waiting for RTO.

Heavy Incast Traffic with Background Workload: We repeat the above experiment, increasing the frequency of mice 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, each server requests a 1.15MB file partitioned into 100 11.5KB chunks totalling $\approx 145MB$ per epoch and a total of $\approx 1.3GB$ for all 9 epochs. As shown in Fig. 9a, even with the increased incast frequency, HSCC scales well despite mice flows having to also compete against bloated elephant flows. Mice flows' average and standard deviation of FCT see similar improvement as the previous experiments compared to TCP with DropTail and DCTCP. This can be attributed to the decreased packet drops rate with the help of HSCC and hence lesser timeouts are experienced as shown in Fig. 9c. Compared to the previous experiment, Fig. 9b shows, elephants throughput is reduced because of frequent rate throttling introduced by HSCC during incast periods. However, we believe that the bandwidth is fairly utilized by mice and elephants with HSCC, hence the lower elephant goodput when mice are active.

VIII. Related Work

Numerous proposals have been devoted to addressing congestion problems in data center networks (DCNs) and in particular incast congestion. Recent works [28–30] 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 windows in TCP's congestion control.

Towards solving the incast problem, one of the first works [31] 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 [7] suggested modifying the TCP protocol in data centers by reducing the value of the minRTO value from 200ms to microseconds scale. Then DCTCP [3] and ICTCP [32] 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 to handle incast traffic by adjusting the TCP receiver window proactively, before packets are dropped. However, all these solutions require changing the TCP protocols at the end users, they can not react fast enough with the dynamic nature of data center traffic and they impose a limit on the number of senders.

Similar to DCTCP, DCQCN [33] 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 end-points. TIMELY [5] is another congestion control mechanism for data centers which tracks fine-grained sub-microsecond updates in RTT as network congestion indication. Timely's fine-grained tracking may increase CPU utilization of the end hosts and is sensitive to traffic variations in the backward path.

IX. Conclusions and Future Work

In this paper, we showed empirically that the low bandwidth-delay product of data centers results in short loss cycles for TCP. Because of such short loss cycles, the predominant short-flows in data center turn out to experience frequent non-recoverable losses, inflating thereby their FCT by the TCP minRTO which is several orders of magnitude larger than the RTT. To improve TCP performance in data center we proposed to stretch the TCP cycles via a hysteresis controller that alternates between TCP when congestion is mild and a conservative CBR rate when congestion is likely to occur. Using control theory we showed that our controller is stable. Using ns2 simulation we studied the performance and sensitivity of our switch algorithm to parameter setting and finally using a combination of Verilog and Linux Kernel programming we built a prototype of the HSCC switch onto the NetFPGA platform. We deployed our prototype in our small scale experimental data center and demonstrated via experimental results that our switching algorithm indeed improves the average FCT, the FCT variance and the tail FCT of small flows which are known to predominate in data centers. As part of future work, we are testing the algorithm in a much larger data center with higher speeds and investigating an end-host scheme that uses ECN feedback to perform the switching.

References

[1] S. Ha and I. Rhee, "CUBIC : A New TCP-Friendly High-Speed TCP Variant," *ACM SIGOPS Operating Systems Review*, vol. 42, 2008.

[2] D. X. Wei, C. Jin, S. H. Low, and S. Hegde, "FAST TCP: Motivation, Architecture, Algorithms, Performance," *IEEE/ACM Transactions on Networking*, vol. 14, no. 6, pp. 1246–1259, 2006.

[3] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data center TCP (DCTCP)," *ACM SIGCOMM CCR*, vol. 40, p. 63, 2010.

[4] M. Alizadeh, A. Kabbani, B. Atikoglu, and B. Prabhakar, "Stability analysis of QCN," *ACM SIGMETRICS*, vol. 39, no. 1, p. 49, 2011.

[5] 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 *ACM SIGCOMM*, 2015.

[6] M. Mellia, I. Stoica, and H. Zhang, "TCP Model for Short Lived Flows," *IEEE COMMUNICATIONS LETTERS*, vol. 6, no. 2, 2002.

[7] V. Vasudevan, A. Phanishayee, H. Shah, E. Krevat, D. G. Andersen, G. R. Ganger, G. A. Gibson, and B. Mueller, "Safe and effective fine-grained TCP retransmissions for datacenter communication," *ACM SIGCOMM Computer Communication Review*, vol. 39, p. 303, 2009.

[8] M. Alizadeh, S. Yang, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker, "Deconstructing datacenter packet transport," *Proceedings of the 11th ACM HotNets workshop*, 2012.

[9] Wei, L. Chen, K. Chen, D. Han, C. Tian, and H. Wang, "Information-agnostic flow scheduling for commodity data centers," in *Proceedings of the 12th USENIX NSDI*, 2015.

[10] A. Shieh, S. Kandula, A. Greenberg, C. Kim, and B. Saha, "Sharing the data center network," in *Proceedings of USENIX NSDI*, 2011.

[11] E. Zahavi, A. Shpiner, O. Rottenstreich, A. Kolodny, and I. Keslassy, "Links as a Service (LaaS)," in *in Proceedings of IEEE ANCS*, 2016.

[12] D. Xie, N. Ding, Y. C. Hu, and R. Kompella, "The only constant is change: Incorporating time-varying network reservations in data centers," in *Proceedings of ACM SIGCOMM*, 2012.

[13] D. Zats, T. Das, P. Mohan, D. Borthakur, and R. Katz, "DeTail : Reducing the Flow Completion Time Tail in Datacenter Networks," in *ACM SIGCOMM*, pp. 139–150, 2012.

[14] W. Bai, L. Chen, K. Chen, and H. Wu, "Enabling ECN in multi-service multi-queue data centers," *Proceedings of USENIX NSDI*, 2016.

[15] B. Cronkite-Ratcliff, A. Bergman, S. Vargaftik, M. Ravi, N. McKeown, I. Abraham, and I. Keslassy, "Virtualized congestion control," in *Proceedings of SIGCOMM*, 2016.

[16] K. He, E. Rozner, K. Agarwal, Y. J. Gu, W. Felter, J. Carter, and A. Akella, "Ac/dc tcp: Virtual congestion control enforcement for datacenter networks," in *Proceedings of SIGCOMM*, 2016.

[17] M. Mathis, J. Semke, J. Mahdavi, and T. Ott, "The macroscopic behavior of the TCP congestion avoidance algorithm," *ACM Computer Communication Review*, vol. 27, pp. 67–82, 1997.

[18] B. A. Greenberg, J. R. Hamilton, S. Kandula, C. Kim, P. Lahiri, A. Maltz, P. Patel, S. Sengupta, A. Greenberg, N. Jain, and D. A. Maltz, "VL2: a scalable and flexible data center network," in *Proceedings of ACM SIGCOMM*, 2009.

[19] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken, "The nature of data center traffic," in *Proceedings of IMC*, 2009.

[20] T. Benson, A. Anand, A. Akella, and M. Zhang, "Understanding data center traffic characteristics," *SIGCOMM CCR*, vol. 40, pp. 92–99, 2010.

[21] V. Jackbson, R. Braden, and D. Borman, "TCP Extensions for High Performance," 1992. https://www.ietf.org/rfc/rfc1323.txt.

[22] 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," *ACM Computer Communication Review*, vol. 30, 2000.

[23] H. K. Khalil, *Nonlinear systems*. Prentice Hall, 1996.

[24] M. Alizadeh, "Data Center TCP (DCTCP)." http://simula.stanford.edu/ alizade/Site/DCTCP.html.

[25] iperf, "The TCP/UDP Bandwidth Measurement Tool." https://iperf.fr/.

[26] Apache.org, "Apache HTTP server benchmarking tool." http://httpd.apache.org/docs/2.2/programs/ab.html.

[27] OpenvSwitch.org, "Open Virtual Switch project." http://openvswitch.org/.

[28] Y. Chen, R. Griffith, J. Liu, R. H. Katz, and A. D. Joseph, "Understanding TCP incast throughput collapse in datacenter networks," in *Research on Enterprise Networking Workshop (WREN)*, 2009.

[29] J. Zhang, F. Ren, and C. Lin, "Modeling and understanding TCP incast in data center networks," in *IEEE INFOCOM*, 2011.

[30] W. Chen, F. Ren, J. Xie, C. Lin, K. Yin, and F. Baker, "Comprehensive understanding of TCP Incast problem," in *INFOCOM*, 2015.

[31] E. Krevat, V. Vasudevan, A. Phanishayee, D. G. Andersen, G. R. Ganger, G. A. Gibson, and S. Seshan, "On application-level approaches to avoiding TCP throughput collapse in cluster-based storage systems," in *Proceedings of Supercomputing - PDSW*, 2007.

[32] H. Wu, Z. Feng, C. Guo, and Y. Zhang, "ICTCP: Incast congestion control for TCP in data-center networks," *IEEE/ACM Transactions on Networking*, vol. 21, 2013.

[33] Y. Zhu, H. Eran, D. Firestone, C. Guo, M. Lipshteyn, Y. Liron, J. Padhye, S. Raindel, M. H. Yahia, and M. Zhang, "Congestion control for large-scale rdma deployments," in *Proceedings of the ACM SIGCOMM*, 2015.