Control Theory Based Hysteresis Switch for Congestion Control in Data Centers

Ahmed M. Abdelmoniem and Brahim Bensaou
Department of Computer Science and Engineering
The Hong Kong University of Science and Technology
Clear Water Bay, Hong Kong
{amas, brahim}@cse.ust.hk

June 28, 2017

Abstract

n this report, we empirically and analytically study TCP packet loss cycles which greatly affect the performance of TCP applications. Specifically, the short cycles of TCP in data center environments degrade the Flow Completion Time (FCT) of most time-sensitive (e.g., partition-aggregate) TCP applications. In particular, for small flows, short loss cycles may lead to losses only recoverable by Retransmission-Timeout (RTO) which expands the FCT by 2-4 orders of magnitude. We find that short loss cycles are merely the symptom of a pathology of TCP's AIMD algorithm design targeting moderate-bandwidth high-delay Internet and is shown to be inadequate for the high-bandwidth low-delay data centers. Hence, the common sense calls for an alternative methodology that can effectively expand the TCP loss cycle long enough for short flows to finish the transfer within one or fewer RTTs. To this end, in this report, we propose a switch-based control which synthesizes Hysteresis Switching Congestion Control (HSCC). HSCC switches between TCP and a slow CBR in order to expand the TCP loss cycle within data centers. The proposed scheme is studied analytically then it is evaluated via means of small and large scale NS2 simulation and real experimentation in a real testbed cluster. The results show considerable improvements in the FCT distribution and the missed deadlines. HSCC

achieve this without modifying or interfering with the VM's TCP implementation or configuration. This makes the scheme very appealing for deployment in public data centers.

1 Introduction

We have shown that TCP is still the dominant transport protocol in use by most cloud application. TCP is a distributed end-to-end protocol that relies on a collections of algorithms to achieve reliable and effective communication. Mostly, these algorithms were not part of the initial incarnation of TCP and they were added gradually to parallel the evolution of the Internet. Therefore it is not surprising that most TCP implementations found in the most popular operating systems today are by default fine-tuned to be efficient in the Internet. In particular, one algorithm that affects the performance of the applications dramatically is the TCP congestion control mechanism: over the years numerous variants of the TCP congestion control mechanism have seen the light, mostly to meet the ever evolving design goals and operational requirements of new operating environments [11, 7, 13, 30, 28]. To name a few examples, TCP Westwood [7] was proposed to improve TCP performance in lossy wireless networks, Cubic TCP was designed to take advantage of high-speed long-distance networks and Fast TCP [30] is a delay-based variant designed to improves TCP throughput over high-speed long-distance networks.

In the same spirit, small short-lived flows are observed to experience unnecessarily long FCT in data centers with vanilla-TCP. As a result several new variants of TCP congestion control have been proposed to improve the performance of TCP in data center networks (e.g., DCTCP [1, 2], TIMELY [25]). In contrast, other studies simply identified the sources of performance degradation in data centers with vanilla-TCP and proposed tuning congestion control parameters to match the scale of data center networks (e.g., reducing the initial congestion window to cope with the small switch buffers [23] or scaling down the minimum retransmission timeout to match the small RTT of a data center [29]). All these approaches have been shown to yield some performance improvements, and some are already in use in production data centers. However, we see these solutions as only applicable to private data centers where the operator has control over both ends of the internal TCP connections. Several works have investigated switch-based congestion control means to improve the FCT of short-lived flows. For example, pFabirc [3] and PIAS [4] leverage priority queuing in the switches to segregate and serve short-lived flows before long-lived ones. These mechanisms

also apply exclusively to privately owned data centers as they require modifications of the TCP stack of the end-system or the guest VM (e.g., PIAS relies on DCTCP and pFabric relies on a modified version of TCP).

In public, multi-tenanted, datacenters, the tenants lease and share a common physical infrastructure to run their applications on virtual machines (VMs). The tenants can implement and deploy their preferred version of TCP and thus of congestion control algorithm. However, the operators have no control over the guest operating system nor the TCP variant running in the guest VMs. To tackle this issue, few approaches have been proposed in the literature. First, the public datacenter operator can statically apportion the network bandwidth among the tenants, giving each of them a fixed allocation with guaranteed bounds on delays [26, 32]. This technique though effective, would not benefit from the statistical multiplexing resulting in an ineffectively unused admissible region. Another approach suggests modifing all the switches in the datacenter to ensure small buffer occupancies at each switch. This can be achieved by leveraging separate weighted queues and/or applying various marking thresholds within the same queue [21, 27, 5]. Typically, each source algorithm requires a certain weight/threshold to fully utilize the bandwidth. Hence, such schemes are not scalable, may lead to starvation and are hard to deploy due to the increasing number of different congestion control algorithms employed by the tenants.

To enable true deployment potential in such heterogeneous TCP environment without modifying TCP, in this report, we again adopt a switch-based approach. However, to make our scheme agnostic to the nature of the congestion control mechanism employed by the tenant, we rely on a standard universally adopted TCP mechanism to convey congestion signals to the sources. The sources become simple flow rate controllers during congestion while the switches set the source rates during these periods. This approach enables the data center operators to innovate in the switch without paying attention to the TCP variants running in the guest VMs.

In the remainder of this report, we study empirically the impact of TCP RTO on the performance of TCP and then show the relation between the RTO and the TCP cycles in Section 2. The proposed solution, system modeling, design and practical aspects are discussed in Section 3. In Section 5, we present our simulation results in detail. Then, in section 6, we discuss our implementation details and show experimental results from a real deployment in a small-scale testbed. We discuss important related work in Section 7. We finally conclude the report in 8.

For the purposes of reproducibility and openness, we make the code and

scripts of our simulations, implementations and experiments available online at **http://github.com/ahmedcs/HSCC**.

2 Background and Problem Statement

2.1 Timeout Reflection on the FCT

Short-lived flows in data centers face the challenge of operating in small buffered environments while TCP uses inadequate Internet-suited mechanisms such as large initial window, long minRTOs and/or slow-start exponential growth. This inappropriate and composite nature of hardware and TCP configurations frequently leads to devastating timeout events for small flows. In particular, when the number (N) of such flows increases, in the presence of small buffers, they turn out to experience synchronized losses (the so-called TCP incast congestion). Knowing that the loss probability grows linearly with N [22], 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 N flows and let F be the flow size, τ be the mean RTT of the flow and x be the number of RTTs to complete the transfer. Then the optimal throughput is:

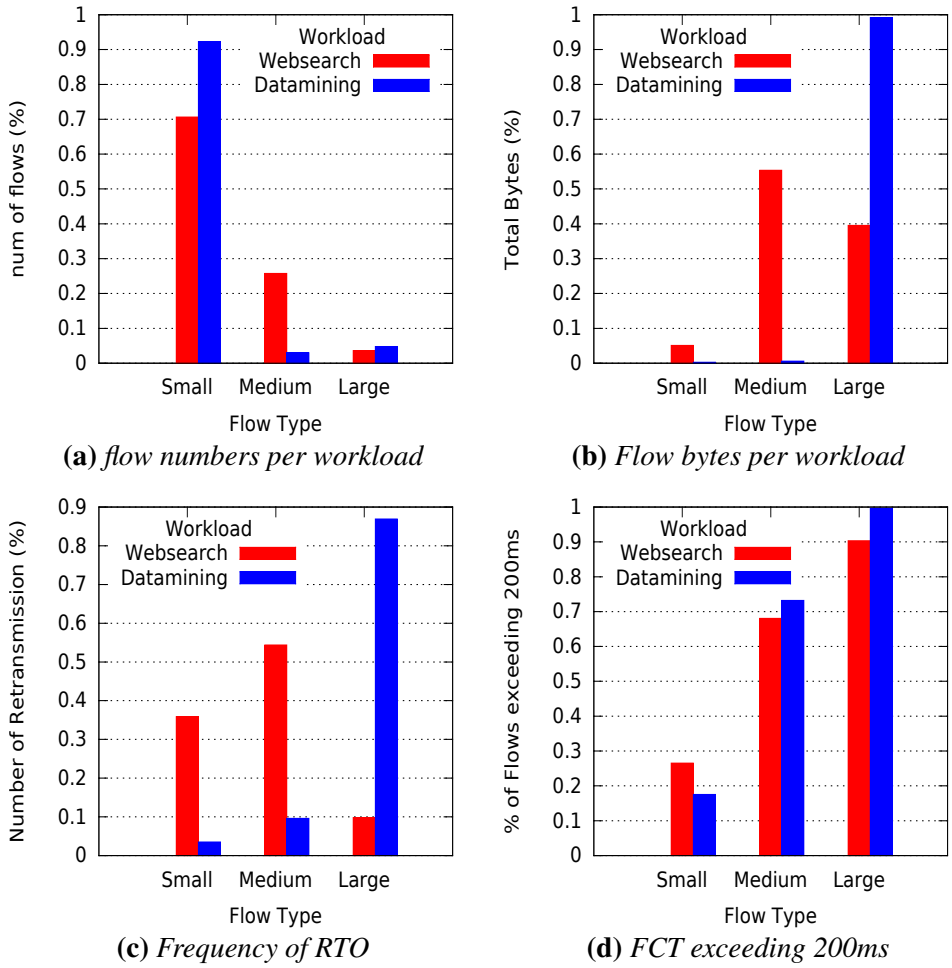
$$\rho* = \frac{F}{x\tau + \frac{NF}{C}} \tag{1}$$

In practice, when TCP incast congestion involving N flows results in throughput-collapse, some flows might experience timeouts and have to recover via RTO. Then the transfer time becomes (RTO + the typical transfer time) and the throughput ρ becomes:

$$\rho = \frac{F}{RTO + x\tau + \frac{NF}{C}} \tag{2}$$

In data centers, the typical RTT is around 100μs, while existing TCP implementations impose a minimum RTO of about 200+ms. For large flows, x is large and hence RTO and $x\tau$ in 2 are comparable. In contrast, for small flows who only last a few RTTs and hence RTO can be at least 2 orders of magnitude larger then $x\tau$. As a consequence, if a small flow experiences a loss that cannot be recovered by 3-DUPACKs (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., $\approx$100ms). Hence, to improve the performance of small flows we recommend curbing NRLs as much as possible.

2.2 Measurement-Based Verification



(a) *flow numbers per workload* **(b)** *Flow bytes per workload*

(c) *Frequency of RTO* **(d)** *FCT exceeding 200ms*

Figure 1: *An experiment to characterize TCP flows and evaluate the effects of RTO on FCT in websearch and datamining workloads.*

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, we built a custom TCP traffic generator. The traffic generator establishes TCP connections to mimic flows with sizes and inter-arrival time distributions drawn from various realistic workloads (e.g., websearch [1] and datamining [12] (as well as others [18, 6]). Then, to track the nature of losses, we collect TCP socket-level events

(e.g., timeouts) via a custom-built Linux kernel module that leverages kernel probing functions (jprobe [17]) and dumps the socket state-variables of interest. The module installs probe objects in each traced TCP function. In our experiments, the probed TCP function is the tcp_retransmit_skb which is called whenever segments are transmitted by TCP. To conduct the experiment, the end-hosts are instrumented with the probe module and a total of 7000 flows are generated. Flows are categorized into small ($\leq 1MB$), medium ($1-10MB$) and large ($\geq 10MB$).

Figure 1a shows the percentage of flows generated from each size while Figure 1b shows the percentage of network bytes generated from each size. We observe that, in websearch and datamining workloads, most flows are small and websearch data bytes are distributed almost uniformly over the three categories. However, in datamining case, most of the bytes are produced by large flows (i.e., these flows tend to be quite large in size). Figure 1c shows the RTO frequency for sequences observed in each category and suggests that RTO is highly likely for all flow types in both workloads. Noticeably, in a websearch workload, RTOs of small flows are $(2563 \approx 35\%)$. Figure 1d shows flows exceeding 200ms are $(\approx 28\%, \approx 18\%)$ for small flows.

As an example, assume on average the small flow size is B=500KB and on average 36 flows [1] share the bottleneck link capacity of C=1Gbps equally. Then, such a flow should finish its transmission in $\approx$ 17 RTTs and the FCT would be $(= \frac{B}{C} + 17100\mu\text{s} = \frac{500\text{KB}}{1\text{Gbps}/(8*36)} + 1.7\text{ms} \approx 150\text{ms})$. According to Figure 1, flows on average would experience 1/2 RTO in websearch (2 in datamining). Then this translates to adding another $\geq$ 100ms ($\geq$ 400ms) to the FCT (i.e., more than $66\%(366\%)$ the ideal FCT for websearch (datamining), respectively. These results show the effect of RTOs on small flows with few segments to send.

The measurements strongly suggest that RTO frequency is non-negligible in data centers for which we suspect that the short TCP loss cycle is to be blamed. In the next section, we analytically explore the anatomy of this problem then propose a possible solution for it.

3 The Proposed Methodology

In this section, we explore the anatomy of the problems discussed previously.

TCP flows start in an exponential increase phase and then enter into Congestion Avoidance (CA). TCP CA is the one that determines the steady-state dynamics of TCP when packet losses are moderate. CA algorithm employs two strategies namely the Additive Increase (AI) and Multiplicative Decrease (MD).

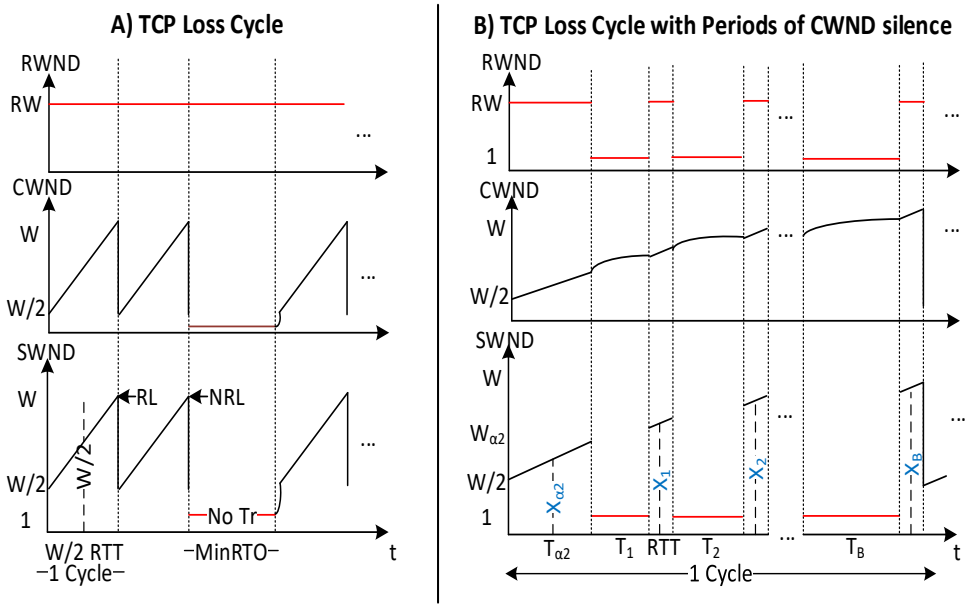


Figure 2: *TCP with DropTail vs TCP with Hystersis-based AQM. Both graphs show the period and the number of MSS sent within each TCP loss cycle.*

AI increases the congestion window by a constant amount (typically, 1 Maximum Segment Size or MSS) in each Round-Trip Time (RTT) upon successful reception of ACKs (i.e., no losses or congestion signals within the current congestion window)[1]. MD decreases the congestion window by a constant multiplicative factor upon arrival of loss or congestion signal[2].

In CA, TCP congestion window ***Cwnd***proceeds in a periodic sawtooth shape as shown in part A of Figure 2. The typical behavior of CA AIMD algorithm ensures ***Cwnd***attain values between the maximum ***Cwnd***(w) and its minimum value (or equilibrium point $\frac{w}{2}$) [22]. The figure also shows that receiver window ***Rwnd***is always (or almost) constant because receivers allocate large and enough buffers. The sending window ***Swnd***is shown to take the minimum of the congestion window ***Cwnd***and the receiver window ***Rwnd***. Typically because of the small RTT in data centers, ***Rwnd*** > ***Cwnd***and the throughput of TCP can be shown to be in-

[1]Typically, in real OS implementations like Linux, this is achieved by counting the number of returned ACKs within 1 RTT and opening the congestion window by 1 MSS at the end of the RTT. This can be approximated by a linear increase of small increments ($\frac{1}{Cwnd}$) for each ACK

[2]Note that, TCP reacts once per RTT for all congestion signals (i.e., N congestion signals do not cut the window N times but only once).

versely proportional to the square root of the loss event probability [22]. As such, because in data centers losses are frequent, the loss cycle turns out to be very short. This problem is found in window-based TCP (e.g., RENO, CUBIC, DCTCP) that rely on loss/congestion signals to adjust their sending rates.

3.1 A Control Theoretic Solution

To increase TCP throughput one needs to stretch the loss cycle. In addition, to discriminate favorably short-lived incast traffic our mechanism makes room for such flows when they are most likely to experience non-recoverable losses, i.e., when the buffer backlog builds up.

Figure 2(A) shows ***Cwnd***, ***Rwnd***and ***Swnd***of TCP cycles with periods where ***Cwnd***is inactive. In each cycle, ***Cwnd***becomes inactive for a certain time and ***Rwnd***takes over the control of ***Swnd***. During this period, ***Rwnd***is set to a slow rate for until ***Cwnd***is reactivated again. In the following, we derive an expression of the data workload transferred in the new loss cycle. While Figure 2(B) shows that ***Cwnd***opens up by 1 MSS in each RTT until w_{α_2} is obtained. It takes $T_{\alpha_2} = w_{\alpha_2} - \frac{w}{2}$ RTTs to go from $\frac{w}{2}$ to w_{α_2}. The amount of data transferred within this period is the area under that part which can be expressed as:

$$D_{\alpha_2} = T_{\alpha_2}\frac{w}{2} + \frac{T_{\alpha_2}}{2}(w_{\alpha_2} - \frac{w}{2}) = \frac{w^2}{8}\frac{{w_{\alpha_2}}^2}{2}, \tag{3}$$

Now the amount of data D transferred within each CBR mode period $T_{i,CBR}$ is the area under the rectangle (i.e., $D_{i,CBR} = 1 \times T_{i,CBR}$). The data transferred within each activation of AI mode for a period of one RTT in the round i can calculated as $D_{(i,AI)} = w_{\alpha_2} + i$. To find the total amount of data transferred until ***Cwnd***reaches the maximum of w, we sum up these terms:

$$\begin{aligned} D_{New} &= D_{\alpha_2} + \sum_{i=1}^{w-w_{\alpha_2}} D_{i,CBR} + D_{i,AI} \\ &= \frac{w^2}{8}\frac{{w_{\alpha_2}}^2}{2} + \sum_{i=1}^{w-w_{\alpha_2}} (T_i + w_{\alpha_2} + i), \end{aligned} \tag{4}$$

The summation term of the right hand side is an arithmetic series, hence the value of the data transferred can be simplified:

$$D_{New} = D_{TCP} + \sum_{i=1}^{w-w_{\alpha_2}} T_i, \tag{5}$$

Figure 2 shows an explanation of the targeted TCP behavior where TCP loss cycle is expanded to cover more RTT rounds. In this case, we assume that TCP can alternate between two sending rates where one is governed by the typical AIMD and the other one is a Constant Bit Rate (CBR) operating at a slow rate of 1 MSS/RTT. As shown in Figure 2, the resulting behavior within 1 loss cycle in CA mode is:

1. TCP start ramping up in the AI mode until a certain congestion window w_{α_2} is reached based on signals coming from the bottleneck switch.
2. When ***Cwnd***hits the w_{α_2} mark, TCP switches to the slow CBR mode sending at rate 1 MSS/RTT until another signal from the switch comes that lets TCP resume TCP AI.
3. Afterwards, TCP will operate in AI mode for a single RTT before switching back to the slow CBR mode, this is mainly due to the discrete nature of TCP senders.
4. This cycle of switching back and forth between AI and slow CBR continues for a few rounds until the ***Cwnd***reaches or exceeds the maximum congestion window w.
5. Then packet loss occurs and MD is applied to restart a new stretched loss cycle.

A toy scenario involving incast event: Figure 3 shows a toy scenario where TCP flow is forced to shares the network with new incoming 5 incast TCP flows. The TCP flow is assumed to be in its steady state (i.e., the maximum w is the value that leads to buffer overflow). Initial congestion window is assumed to be 1 MSS. In Figure 3(A), at time T, 5 flows arrive each with 1 MSS. However, the current TCP flow would continue with its CA and the current sending window is $w-1$. Then, there is room for 1 MSS to be filled by 1 new flow and the other 4 would experience losses. On the other hand, Figure 3(B) shows that. at time T, the hysteresis switch would be active and the current sending window is 1 MSS. Then, clearly there would be room for 1 MSS from each flow including the 5 new arrivals and no flow would experience losses. The performance gains are obvious in terms of the completion time for incast flows.

3.2 HSCC System Modeling

HSCC system control loop is depicted in Figure 4a, the system consists of four main components namely three data sources, the queue and a hysteresis controller

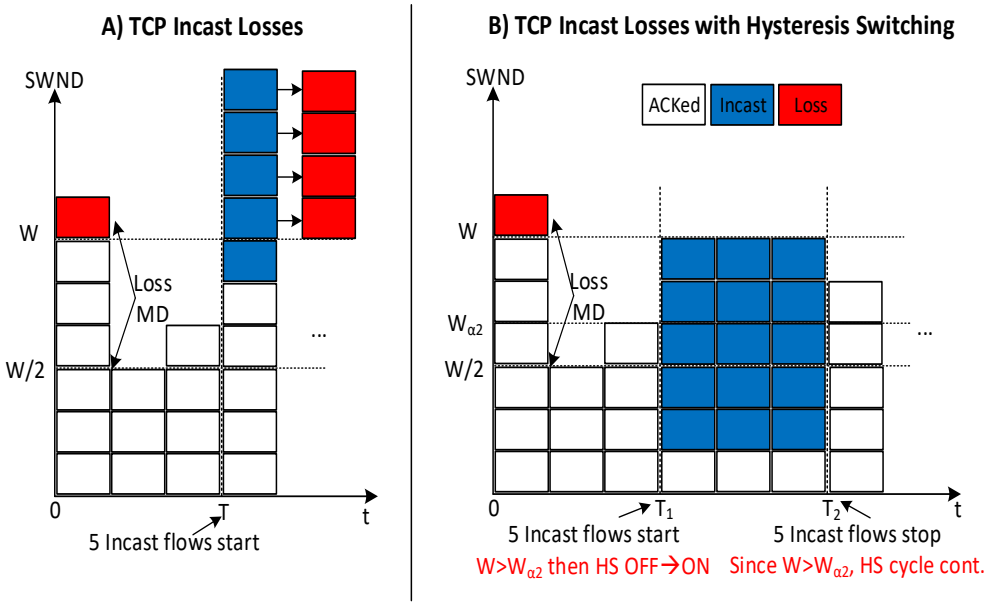
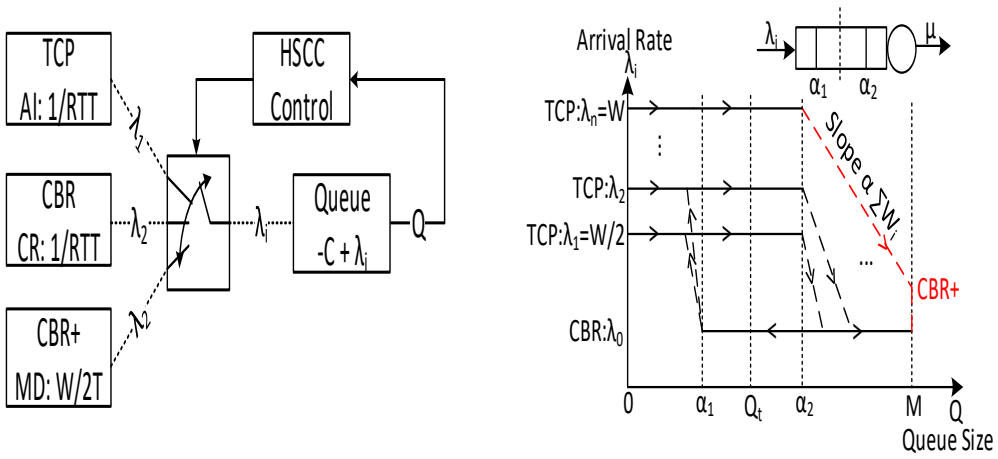


Figure 3: *TCP with DropTail vs TCP with Hysteresis Switched AQM. Both graphs show the evolution of the congestion window for the active flow and the new segments introduced by the incast traffic.*

that switches among the three data sources. These include a TCP Additive increase source with 1 MSS/RTT increase rate, a CBR source sending at 1 MSS/RTT and a CBR+ source that combines the CBR source of 1 MSS/RTT and a Multiplicative Decrease on the congestion window. Figure 4b shows the switching control law used by the HSCC switch, it is a Counter-Clockwise (CC) hysteresis system. In CC hysteresis, the switching happens first when the high threshold α_2 is crossed and the state continues until the lower threshold α_1 is crossed. The possible sequence of switching is as follows: *i)* while using TCP source sending at rate λ_1 and the high threshold α_2 is hit then the controller switches to the lower rate CBR source with rate λ_2; *ii)* The system keeps operating in this CBR mode with rate λ_2 until the lower threshold L is hit at which point the controller switches to TCP source to recover the sending rate λ_1; *iii)* There is the special case of switching in dynamics whenever the queue exceeds the buffer space (i.e., M) leading to the loss of packets. In such case, the system switches to a third system CBR+ sending at rate λ_2 as well as applying multiplicative decreasing the rate λ_1 (i.e., TCP congestion window) before crossing back the lower threshold *iv)* otherwise, the system stays in the current state.



(a) *A schematic diagram of the system* **(b)** *HSCC hysteric control low*

Figure 4: *(a) HSCC system components and the feedback loop shows the hysteresis law controlling the switch 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*

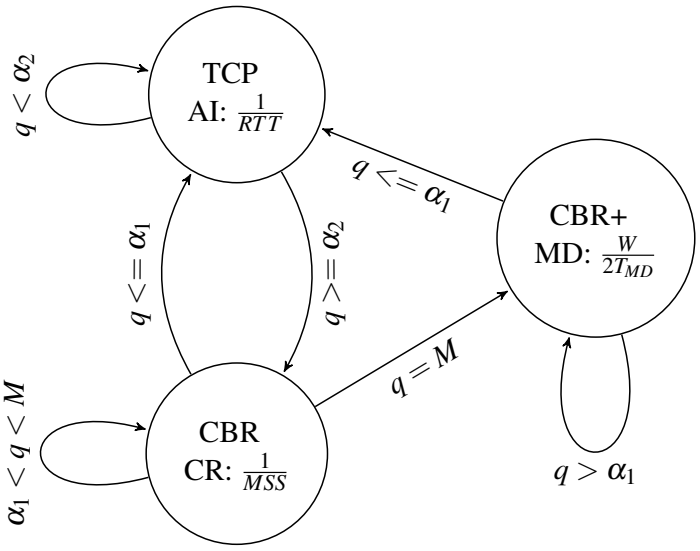


Figure 5: *Flow chart depicting the state transition space of HSCC system*

Figure5 shows the transition diagram of TCP-HSCC system states. It is clear that there is a singularity in the congestion window value after 1 RTT from loss event which is cut by half. These events are caused by the queue length exceeding the buffer size $q >= M$ as shown in the diagram. However, in loss events, the HSCC control law will immediately switch to the CBR source (i.e., CBR+ in Figure 4). Then, it is clear that from the transition diagram that we are operating with CBR until the queue falls back to the low threshold which reactive TCP source again. This means there is a time-period in which the new congestion window $\frac{w}{2}$ is inactive (i.e., the period needed for the queue to drain from being full back to the low threshold).

3.3 HSCC Switch System Design

Figure 6 shows the HSCC system components and operations. First, at connection-setup, flows are hashed into a hash-table with the flow's 4-tuples (i.e., source IP, dest. IP, source port and dest. port) used as the key and the window scale factor used as the value. Flow entries are cleared from the table when a connection is closed (i.e., FIN is sent out). The module writes the scale factor for all outgoing ACK packets in the 4-bit reserved field of TCP headers (alternatively, by using 4-bits of the receive window field and using the remaining 12 bits for window values). The used reserved bits is cleared after their usage by the HSCC switch to avoid packet being dropped by destination due to invalid TCP check-sum value which avoids the need for recalculating TCP checksum at the end-host and the switch. As shown in Figure 6, the module resides right above the NIC driver for a non-virtualized setups and right below the hypervisor to support VMs in cloud datacenters. Hence, this placement does not touch any network stack implementation of host or guest OS, making it readily deployable in production datacenters. 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 whenever it detects the onset of possible incast event for one of the ports, it immediately switches to incast mode and will start window updates in the incoming ACKs.

Control Packets Loss: TCP packets are 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 timeouts. 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

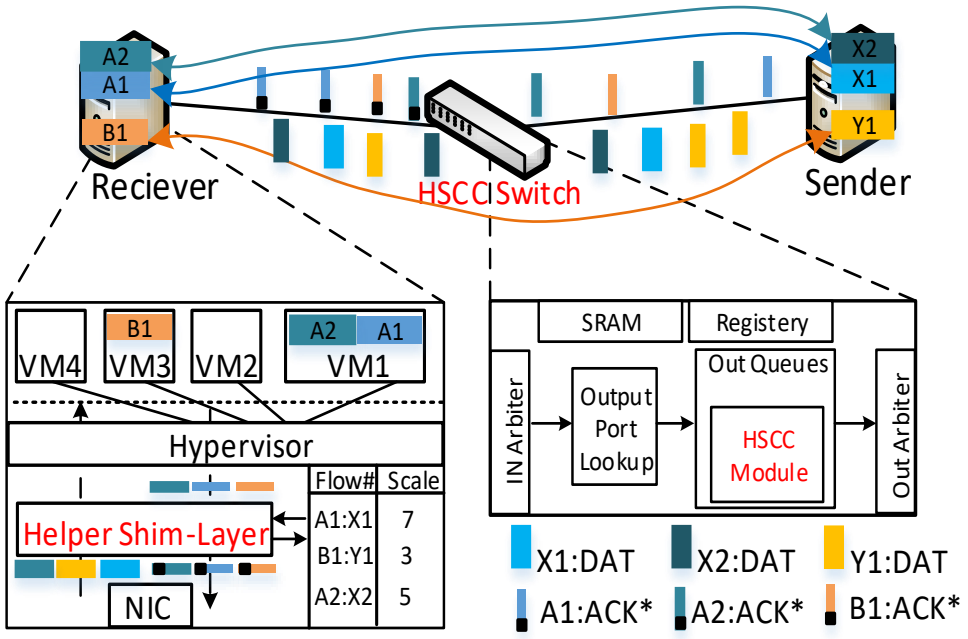


Figure 6: *HSCC System: It consists of an end-host module that attaches scaling value to ACKs and HSCC switch which performs Hysteresis switching between TCP and CBR.*

queues below the size of the buffer. σ is set to reserve a small amount of buffer (e.g., $\approx$3%) for different control packets like pure ACK packets. If ACKs are piggy-packed, then similar to [10], the switch can cut the payload and forward only the ACK. The evaluation of these tweaks is left for future work.

Receive Window Scaling: HSCC relies on a scale factor to rescale the modified window written into TCP header of incoming ACKs. TCP specification [16] 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 let its peer know what factor it uses for its own window value scaling. TCP implementations in most popular Operating Systems including linux adopt the latter approach to save overhead and wasted bandwidth of the former approach. The scaling may be unnecessary for networks with Bandwidth-Delay Product (BDP) 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., 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 in data centers are extremely small [1]. HSCC should be designed to handle window scaling via flow-level tracking at the switch. However, this would make HSCC undesirable and hence we propose a light-weight end-host shim-layer to explicitly send scaling factor with outgoing ACKs. The shim-layer extracts and stores from outgoing SYN and SYN-ACK packets the advertised scaling factor (i.e., within the window scaling option) for each established TCP flow. The shim-layer encodes the scale factor using 4 of the 8 reserved bits of TCP header. Then, the switch uses this value to scale the new window properly and then clear the reserved bits. In ethernet networks, IP checksum is not checked by forwarding devices and checked at the receiver IP layer. So by clearing the reserved bits, computation of new IP checksum is avoided both at the shim-layer and the switch.

4 Mathematical Modeling and Stability

In this section, we will drive the stability analysis using the standard fluid modeling and linearization methods.

Figure 5 shows the state transitions in TCP-HSCC where the control switches between two systems namely AIMD of TCP and slow CBR when HSCC is actively rewriting ***Rwnd***. We model TCP-HSCC behavior, three decoupled sets of differential equations representing the dynamics each system. We follow same modeling procedure in [24, 14]. In our model, we assume a finite buffering capacity M, the RTT differences among competing flows are negligible, packets size P is constant and sources have constant supply of data (i.e., long-lived flows). Note that, mice flows are considered as a temporal low-frequency disturbance/noise imposed on the system and absorbed by the system dynamics. The RTT for a packet P with bottleneck link capacity of C is $\tau_i(t) = T_c + T_t + T_p + \frac{q_i(t)}{C}$, where $T_t = \frac{size(P)}{C}$ 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. Let $P_l \in [0,1]$ be the drop probability triggering 3-DUPACK recovery and $P_u \in [0,1$ be the ***Rwnd***update probability when hysteresis switch is ON (i.e., CBR is active).

The differential equations of TCP AIMD is:

$$\begin{aligned}\frac{dw(t)}{dt} &= \left(\frac{1-P_u(t-\tau_i(t))}{\tau_i(t)} - \frac{w_i(t)P_l(t-\tau_i(t))}{2}\right)\frac{w(t-\tau(t))}{\tau(t-\tau(t))},\\ \frac{dq(t)}{dt} &= (1-P_u(t-\tau_i(t))\frac{\sum w_i(t)}{\tau_i(t)} - C,\end{aligned} \tag{6}$$

Then, the equations in CBR mode can be written as:

$$\begin{aligned}\frac{dw(t)}{dt} &= \left(\frac{P_u(t-\tau_i(t))}{w_i(t)\tau_i(t)}\right)\frac{w_i(t-\tau_i(t))}{\tau_i(t-\tau_i(t))},\\ \frac{dq(t)}{dt} &= P_u(t-\tau_i(t))\frac{\sum 1}{\tau_i(t)} - C,\end{aligned} \tag{7}$$

In the above formulations, $w_i(t)$ and $\tau_i(t)$ should converge in steady state for the competing TCP flows due to the negligible RTT difference assumption. N is considered the long-run average of the number of TCP flows contributing the most to the queue (i.e., elephants). It is intuitive to assume that the queue length q and window size w variables are positive and bounded. Then, queue and window size are assumed to in $q \in [0,M]$ and $w \in [0,W_{max}]$ where M is the buffer size and W_{max} is the maximum window, respectively. To perform the stability analysis the non-linear system of equations is linearized around operating point using the linearization methods in [19, 14]. The combined differential equations regulating the whole TCP-HSCC system dynamics can be formulated as follows:

$$\begin{aligned}\frac{dw(t)}{dt} &= \left(\frac{1-P_u(t-\tau(t))}{\tau(t)} + \frac{P_u(t-\tau(t)}{w(t)\tau(t)} - \frac{w(t)P_l(t-\tau(t)}{2}\right)\frac{w(t-\tau(t))}{\tau(t-\tau(t))},\\ \frac{dq(t)}{dt} &= (1-P_u(t-\tau(t))N\frac{w(t)}{\tau(t)} + P_u(t-\tau(t))N\frac{1}{\tau(t)} - C.\end{aligned} \tag{8}$$

4.1 Linearization of Non-Linear Fluid Model

In the system of equations (8), the system state is defined by the pair $(w(t),q(t))$ and has two inputs (P_l,P_u). Since by definition of the FIFO queue, it is straightforward to notice that if $\alpha_2 \leq M$ and let $P = P_u$ then $P \geq P_l$ and $P_l = \kappa P$ and where $\kappa \in [0,1]$. $\kappa =$ is a positive scalar representing the number of queue passages over the high threshold α_2 before a single buffer overflow. Since after every passage through the high α_2 threshold, when the TCP mode becomes active the

window increases by 1 MSS per flow and if N is given then κ can be calculated. Hence, κ is the number of packets between the high threshold $\alpha_2 M$ 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 $(w_0, q_0, P_{u0} = P_0, P_{l0} = \kappa P_0)$. Hence, the equilibrium points can be found by solving Equations (8) for $\frac{dw}{dt} = 0$ and $\frac{dq}{dt} = 0$ as follows:

$$\begin{aligned}
\frac{dw}{dt} &= \left(\frac{1-P_0}{\tau_0} + \frac{P_0}{w_0\tau_0} - \frac{w_0\kappa P_0}{2}\right)\frac{w_0}{\tau_0} = 0 \\
0 &= \frac{1}{\tau_0} - \left(\frac{P_0}{\tau_0} - \frac{P_0}{w_0\tau_0} + \frac{w_0\kappa P_0}{2}\right) \\
\frac{1}{\tau_0} &= \left(1 - \frac{1}{w_0} + \frac{\kappa w_0\tau_0}{2}\right)\frac{P_0}{\tau_0} \\
P_0 &= \left(\frac{\kappa w_0\tau_0}{2} - \frac{1}{w_0} + 1\right)^{-1}
\end{aligned} \tag{9}$$

$$\begin{aligned}
\frac{dq}{dt} &= \frac{Nw_0}{\tau_0}(1-P_0) + \frac{N}{\tau_0}P_0 - C = 0, \\
\frac{C\tau_0}{N} &= (1-P_0)w_0 - P_0 \rightarrow w_0 = \frac{C\tau_0/N - P_0}{1-P_0}
\end{aligned} \tag{10}$$

To sum up, the equilibrium points P_0 and w_0 are defined as follows:

$$\begin{aligned}
\frac{dw(t)}{dt} = 0 \quad &\rightarrow \quad P_0 = \left(\frac{\kappa w_0\tau_0}{2} - \frac{1}{w_0} + 1\right)^{-1} \\
\frac{dq(t)}{dt} = 0 \quad &\rightarrow \quad w_0 = \frac{C\tau_0/N - P_0}{1-P_0}, \\
&\textit{where} \quad \tau_0 = \frac{q_0}{C} + T.
\end{aligned} \tag{11}$$

We use the obtained equilibrium points, then we define the perturbed variables as $\delta w = w - w_0$, $\delta q = q - q_0$ and the perturbed system input $\delta P_u = P_u - P_{u0}$. From Equation 8, we define two functions $F(w(t), w(t-\tau), q(t), q(t-\tau), P_u(t-\tau)$ and $G(w(t), q(t), q(t-\tau), P_u(t-\tau)$. To linearize the system around the perturbed variables we need to find the partial derivatives of F and G with respect to each of

their parameters and substituting the equilibrium points for the values as follows:

$$
\begin{aligned}
&F(w(t),w(t-\tau),q(t),q(t-\tau),P(t-\tau)) = \\
&\qquad\qquad \left(\frac{(1-P(t-\tau))}{\frac{q(t)}{C}+T}+\frac{P(t-\tau)}{w(t)(\frac{q(t)}{C}+T)}-\frac{w(t)\kappa P(t-\tau)}{2}\right)\frac{w(t-\tau)}{\frac{q(t-\tau)}{C}+T} \\
&G(w(t),q(t),P(t-\tau)) = (1-P(t-\tau))\frac{Nw(t)}{\frac{q(t)}{C}+T}+P(t-\tau(t))\frac{N}{\frac{q(t)}{C}+T}-C.
\end{aligned}
\tag{12}
$$

We first find the partial derivative of F with respect to $w(t)$:

$$
\begin{aligned}
&\frac{\delta F}{\delta w(t)} = \delta\frac{\left(\frac{(1-P(t-\tau))}{\tau(t)}+\frac{P(t-\tau)}{w(t)\tau(t)}-\frac{w(t)\kappa P(t-\tau)}{2}\right)\frac{w(t-\tau)}{\tau(t-\tau)}}{\delta w(t)}, \\
&\frac{\delta F}{\delta w(t)} = \left(-\frac{P(t-\tau(t))}{w(t)^2\tau(t)}-\frac{\kappa P(t-\tau)}{2}\right)\frac{w(t-\tau)}{\tau(t-\tau)}, \\
&\frac{\delta F}{\delta w(t)}|_{w_0,\tau_0,P_0} = \left(-\frac{P_0}{w_0^2\tau_0}-\frac{\kappa P_0}{2}\right)\frac{w_0}{\tau_0} = -\frac{P_0}{\tau_0}\left(\frac{1}{w_0\tau_0}+\frac{\kappa w_0}{2}\right)
\end{aligned}
\tag{13}
$$

Then, we first find the partial derivative of F with respect to $w(t-\tau)$:

$$
\begin{aligned}
&\frac{\delta F}{\delta w(t-\tau)} = \delta\frac{\left(\frac{1-P(t-\tau(t))}{\tau(t)}+\frac{P(t-\tau(t)}{w(t)\tau(t)}-\frac{w(t)\kappa P(t-\tau(t)}{2}\right)\frac{w(t-\tau(t)}{\tau(t-\tau)}}{\delta w(t-\tau)}, \\
&\frac{\delta F}{\delta w(t-\tau)} = \left(\frac{1-P(t-\tau(t))}{\tau(t)}+\frac{P(t-\tau(t)}{w(t)\tau(t)}-\frac{w(t)\kappa P(t-\tau(t)}{2}\right)\frac{1}{\tau(t-\tau)}, \\
&\frac{\delta F}{\delta w(t-\tau)}|_{w_0,\tau_0,P_0} = \left(\frac{1-P_0}{\tau_0}+\frac{P_0}{w_0\tau_0}-\frac{\kappa P_0 w_0}{2}\right)\frac{1}{\tau_0}, \\
&\qquad = \frac{1}{\tau_0^2}-\frac{P_0}{\tau_0^2}\left(1-\frac{1}{w_0}+\frac{\kappa w_0\tau_0}{2}\right) = \frac{1}{\tau_0^2}-\frac{P_0}{\tau_0^2}\frac{1}{P_0} = 0
\end{aligned}
\tag{14}
$$

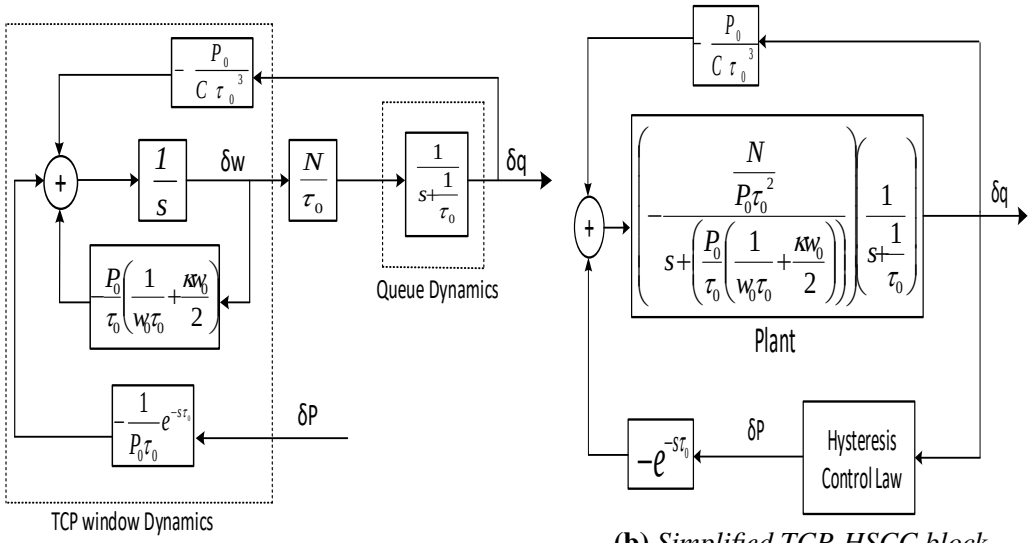
Similarly, we derive the remaining variables and they are:

$$
\begin{aligned}
&\frac{\delta F}{\delta q(t)} = -\frac{P_0}{C\tau_0^3}, \quad \frac{\delta F}{\delta q(t-\tau)} = 0 \\
&\frac{\delta F}{\delta P(t-\tau)} = -\frac{1}{\tau_0 P_0}, \quad \frac{\delta G}{\delta w(t)} = (1-P_0)\frac{N}{\tau_0}, \\
&\frac{\delta G}{\delta q(t)} = -\frac{1}{\tau_0}, \quad \frac{\delta G}{\delta P(t-\tau)} = -(C-\frac{N}{\tau_0}).
\end{aligned}
\tag{15}
$$

Then we can construct the linearized system of equations using the perturbation around the equilibrium point as follows:

$$\begin{aligned}\delta\dot{w}(t) &= -\frac{P_0}{\tau_0}\left(\frac{1}{w_0\tau_0}+\frac{\kappa w_0}{2}\right)\delta w(t) - \frac{P_0}{C\tau_0^3}\delta q(t) - \frac{1}{\tau_0 P_0}\delta P(t-\tau),\\ \delta\dot{q}(t) &= (1-P_0)\frac{N}{\tau_0}\delta w - \frac{1}{\tau_0}\delta q - (C-\frac{N}{\tau_0})\delta P(t-\tau).\end{aligned} \tag{16}$$

By taking the Laplace transform of the equations in 16 and drawing the linearized system components in the block form [19], we get the system shown in Figure 7. Next, we will use the linearized system to derive the stability of TCP-HSCC system.



(a) *Linearized TCP-HSCC block diagram*

(b) *Simplified TCP-HSCC block diagram*

Figure 7: *(a) HSCC system components and the feedback loop shows the input δP, the output δq and the TCP and queue dynamics of the system (b) The simplified block diagram sums up system components into the Plant which takes δP after delay as the input and δq as the output*

4.2 Stability Analysis

Let $C1 = \frac{2P_0}{\tau_0}\left(\frac{1}{w_0\tau_0} + \frac{\kappa w_0}{2}\right), C2 = \frac{P_0}{C\tau_0^3}, C3 = (1-P_0)\frac{N}{\tau_0}$ and $C4 = \frac{1}{\tau_0}$. (C_1, C_2, C_3, C_4) are positive variables. These variables are positive if $w_0 > 0$ and $P_0 > 0$. From Eq. 11, it is easy to see that $w_0 > 0$ if $C\tau_0 \geq N$ and $0 < P_0 < 1$. Condition $P_0 > 0$ means $\frac{1}{w_0} < 1 + \frac{\kappa w_0 \tau_0}{2}$ which is true if $w_0 \geq 1$. The conditions $P_0 < 1$ is true if $0 < \frac{\kappa w_0 \tau_0}{2} < 1$ and $w_0 \geq 1$. So the variables C_1, C_2, C_3 and C_4 are positive if $C\tau_0 \geq N, w_0 \geq 1$ and $0 < \kappa w_0 \tau_0 < 2$

Let the system of equations in matrix form be Z, the system coefficients matrix be A and the input coefficients vector be b, then the system can be expressed as:

$$\begin{aligned} &Z = Ax + by, \quad where \\ &A = \begin{bmatrix} -C_1 & -C_2 \\ C_3 & -C_4 \end{bmatrix} \quad \text{and} \quad x = \begin{bmatrix} \delta w \\ \delta q \end{bmatrix}, \\ &b = \begin{bmatrix} -\frac{1}{\tau_0 P_0} \\ -(C - \frac{N}{\tau_0}) \end{bmatrix} \quad \text{and} \quad y = \delta p(t-\tau). \end{aligned} \tag{17}$$

Then, we can find the eigenvalues of A to evaluate system stability. The eigenvalues are:

$$\lambda_{1,2} = -\frac{C_1}{2} - \frac{C_4}{2} \pm \frac{\sqrt{C_1^2 - 2C_1C_4 + C_4^2 - 4C_2C_3}}{2}. \tag{18}$$

In control systems, the system is stable if its dynamics matrix is a Hurwitz stable. A Hurwitz matrix is a symmetric matrix such that all its eigenvalues lie in the negative part of the real axis (i.e, the open left of the imaginary axis) [19].

The eigenvalues of Equation 18 ($\lambda_{1,2}$) are negative definite if $C_1^2 - 2C_1C_4 + C_4^2 - 4C_2C_3 <= 0$ and hence the system is stable. However, if $C_1 + C_4 < \sqrt{C_1^2 - 2C_1C_4 + C_4^2 - 4C_2C_3}$, one of the eigenvalues is positive and the system becomes unstable. We show that the latter case is infeasible:

$$\begin{aligned} C_1 + C_4 &< \sqrt{C_1^2 - 2C_1C_4 + C_4^2 - 4C_2C_3}, \\ (C_1 + C_4)^2 &< C_1^2 - 2C_1C_4 + C_4^2 - 4C_2C_3, \\ C_1^2 + 2C_1C_4 + C_4^2 &< C_1^2 - 2C_1C_4 + C_4^2 - 4C_2C_3, \\ 4C_1C_4 + C_4^2 &< -4C_2C_3. \end{aligned} \tag{19}$$

Since the condition in the Inequality 19 is infeasible because C_1, C_2, C_3 and C_4 are all positive quantities, then the system is considered stable. Given the values of $w_0, P_0, \tau_0, C, N, \kappa$, we can verify the stability of the system. We give the following numerical examples:

- Consider the following network parameter values of a given system, $w_0 = 10$, $P_0 = 0.5$, $\tau_0 = 0.0001$, $C = 10^9$, $N = 25$, $\kappa = 0.2$. Then, the system matrix and eigenvalues are:

$$A = \begin{bmatrix} -10000.1 & -500 \\ 125000 & -10000 \end{bmatrix} \quad \text{and} \quad \lambda_{1,2} = -10000.05 \pm 7905.7i. \quad (20)$$

  Sine both eigenvalues have a negative real part, the stability of this system is shown.

- if we set $w_0 = 2$ and $N = 1$ then the matrix and eigenvalues become:

$$A = \begin{bmatrix} -2000.5 & -500 \\ 5000 & -10000 \end{bmatrix} \quad \text{and} \quad \lambda_1 = -2326.28, \lambda_2 = -9674.21. \quad (21)$$

  Again both eigenvalues are negative and the stability of the new system is shown.

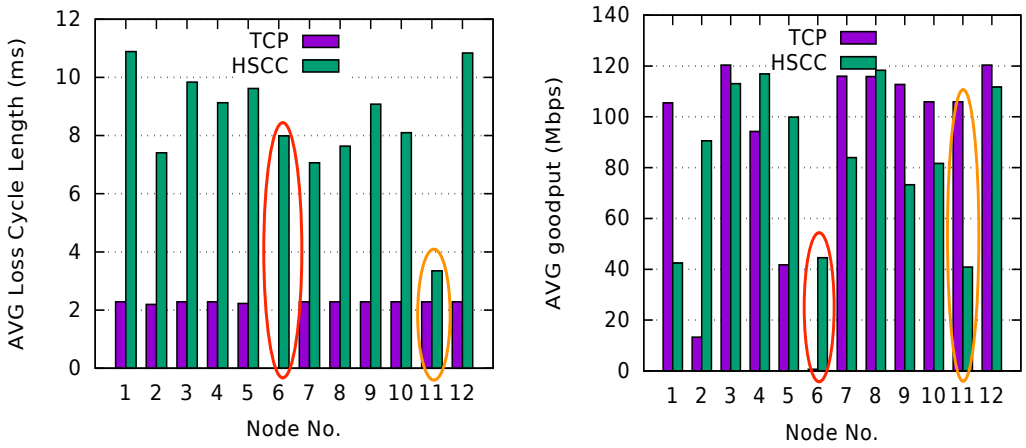
The key observation is that TCP-HSCC system stability is inherited from stability of the original TCP-DropTail system. HSCC mechanism only acts as a temporal interruptions to the original TCP-DropTail controller. These interruptions are meant to expand its loss cycle period to accommodate more packet transfer within each cycle.

5 Simulation Analysis

Here, we conduct a series of simulations in different scenarios and topologies to evaluate HSCC system.

5.1 Microscopic Behavior of HSCC

We first conduct a small-scale simulation in single-rooted tree (or Dumbell) topology where 12 long-lived TCP flows send to a common receiver continuously. We quantify the effect of HSCC on the average length of loss cycle and goodput of



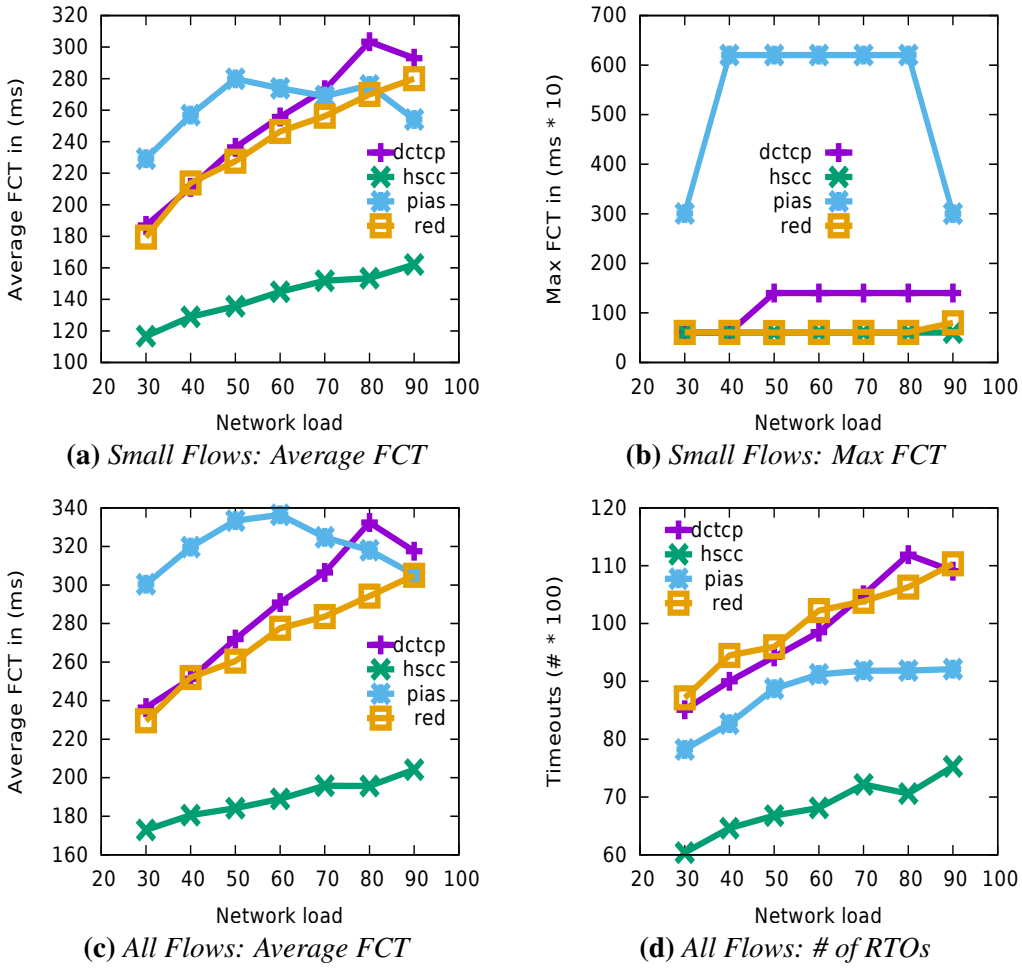
(a) *Average Loss Cylce* **(b)** *Average goodput*

Figure 8: *12 long-lived senders in a Dumbell topology*

each sender. Figure 8a shows the average length of loss cycles in TCP and TCP-HSCC. The results suggest that HSCC can help stretch cycle length by up to 4X. Node 6 (highlighted in red) has zero loss cycle length which was found by inspecting traces is due to starvation (i.e., excessive timeouts). Node 11 (highlighted in orange) achieves little improvement on loss cycle by HSCC. Figure 8b shows the goodput of TCP senders. The results show that HSCC helps flows such as node 6 avoid the starvation and grab enough bandwidth. The goodput of node 11 is degraded due to the small improvement in its loss cycle. To quantify the overall performance, we inspect the Jain's Fairness index (defined as $\frac{(\sum_i X_i)^2}{N\sum_i X_i^2}$). The results indicate that the fairness index with HSCC has improved by 10% from 0.817 to 0.901.

5.2 Realistic Traffic in Datacenter Topology

We conduct simulation analysis of HSCC system in a datacenter-like topology with varying workloads and flow size distributions. We use a spine-leaf topology with 9 leafs and 4 spines using link capacities of 10G for end-hosts and over-subscription ratio of 5 (the typical ratio in current production datacenters is in range of 3-20+). Each ToR hosts 16 server totaling 144 servers and per-port queue is set to 84 packets. We examine scenarios that covers various schemes discussed in Section 2 (e.g., TCP with RED-ECN, DCTCP and PAIS) and compare their



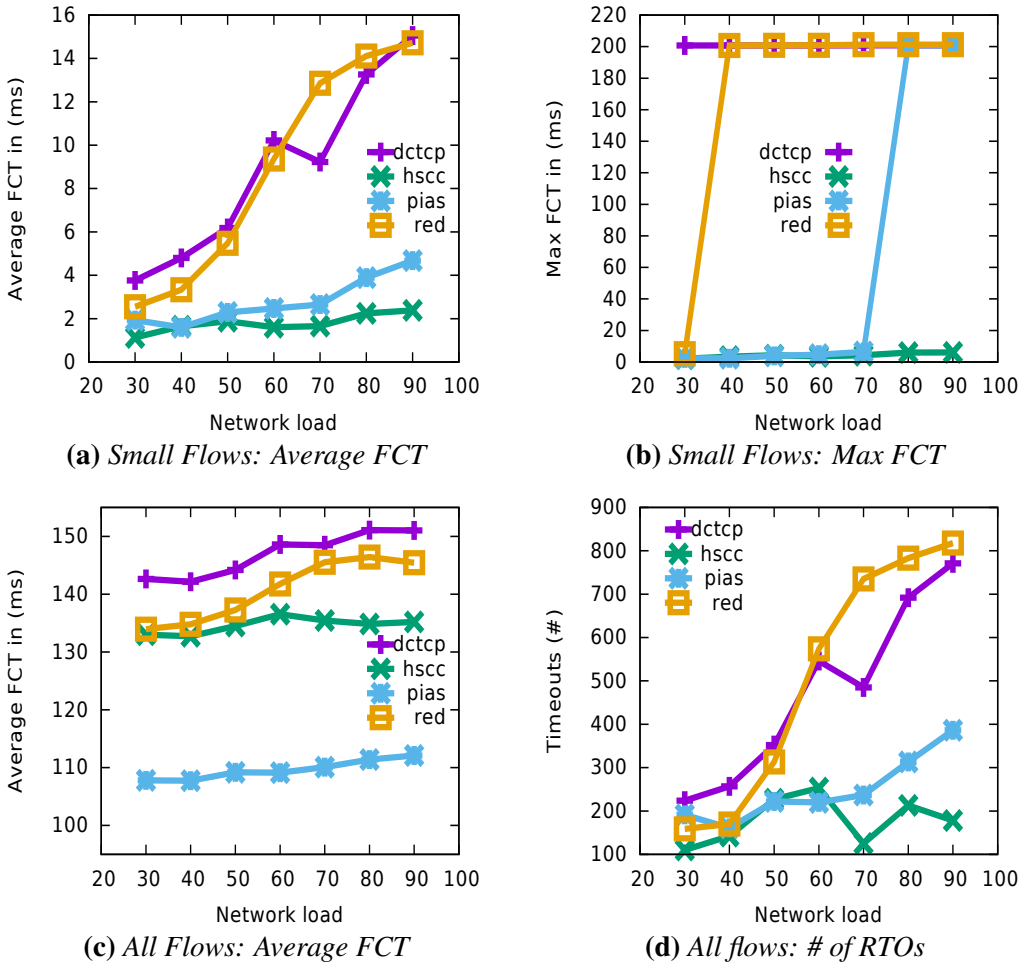
(a) *Small Flows: Average FCT* **(b)** *Small Flows: Max FCT*

(c) *All Flows: Average FCT* **(d)** *All Flows: # of RTOs*

Figure 9: *Performance metrics of simulation runs in a large 9 Leaf - 4 Spine topology using Websearch Workload. The traffic generator varies the network load in the range of [30%, 90%]*

performance with HSCC. The performance metrics of interest are FCT for mice flows (i.e., [0-100] Kbtyes), the average FCT of all flows, # of timeouts and # of unfinished flows. In simulations, per-hop link delays of 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. 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 is set to be equal to the bandwidth-delay product between end-points within one physical rack. Low threshold α_1 is set to 25% and high threshold α_2 is set to 50% for HSCC which can be achieved using a shift-register. To ensure fair comparison with other schemes, control protection feature was disabled.



(a) *Small Flows: Average FCT*

(b) *Small Flows: Max FCT*

(c) *All Flows: Average FCT*

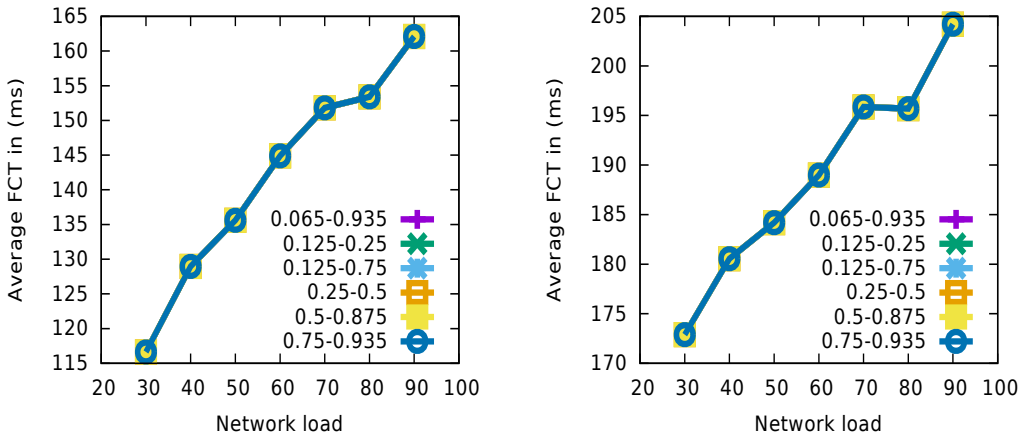
(d) *All flows: # of RTOs*

Figure 10: *Performance metrics of simulation runs in a large 9 Leaf - 4 Spine topology using Datamining Workload. The traffic generator varies the network load in the range of [30%, 90%]) Websearch workload. (e-h) Datamining workload.*

Figure 9 and Figure 10 shows the average and maximum FCT for small flows as well as the average FCT and total timeouts of all flows in websearch and datamining workloads, respectively. The results show the performance of the four

schemes. We observe that HSCC can greatly improve the FCT of small flows on average and tail. 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 average and max FCT. The results suggests that stretching loss cycles can lead significant performance gains. We note that PIAS performs better in datamining workload and worse in websearch workload. We suspect that the fixed 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.

5.3 Sensitivity to the Choice of Thresholds



(a) *Small Flows: AVG FCT* **(b)** *All Flows: AVG FCT*

Figure 11: *Average FCT for small and all flows when $\alpha 1$ and $\alpha 2$ is varied in simulations using websearch.*

We here repeat the last simulation experiment using websearch workload with various values of low threshold $\alpha 1$ and high threshold $\alpha 2$ to assess the sensitivity of HSCC to parameter settings.

The simulation settings, flow sizes, inter-arrival times and network loads are the same as in the previous setup. We report here the achieved FCT of small and all flows in each case. As shown in Figure 11, the FCT is not affected at all by the choice of the parameter $\alpha 1$ and $\alpha 2$. Similar results are observed for dataminig. 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 and the operator can deploy without worrying about the right values for thresholds. Further testing and verification is part of our ongoing work.

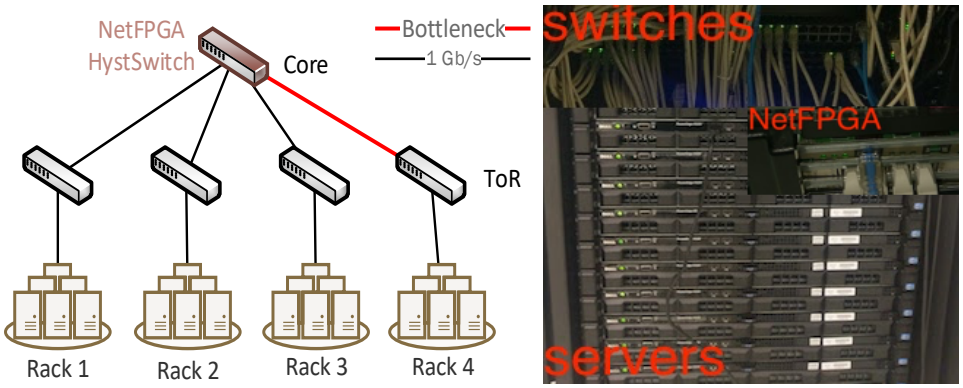
6 Implementation and Experiments

We now investigate the performance of the hardware prototype of HSCC controller namely "HystSwitch" switch as well as its end-host helper module which are presented in Appendix **??**. We prototyped HystSwitch on the NetFPGA platform and used it to conduct a series of testbed experiments to verify its potential.

Figure 6 has shown the deployment of HystSwitch system and the interaction between its components. The HystSwitch system performs the following functions:

1. At connection-setup, flows are hashed into a hash-table with the flow's 4-tuples (i.e., source IP, dest. IP, source port and dest. port) used as the key and the scale used as the value.
2. Flow entries are cleared from the table when a connection is closed (i.e., FIN is sent out).
3. The module writes the scale factor for all outgoing ACK packets in the 4-bit reserved field of TCP headers (or, encode the scale factor into 4-bits of the receive window field and use the remaining 12 bits for the receive window values).
4. 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.
5. The switch module whenever whenever the high threshold is exceeded until it crosses back the low threshold, it continues to update receive window of the incoming ACKs.
6. HystSwitch uses the attached scale factor to rescale the used window so that it can be interpreted correctly by the ACK receiving end-point. Then, it clears the used reserved bits to avoid packet being dropped by destination due to invalid TCP check-sum value which avoids the need for recalculating TCP checksum at the end-host and the switch.

6.1 Experimental Results and Discussion



(a) *Testbed Tree Topology* **(b)** *The Actual Testbed*

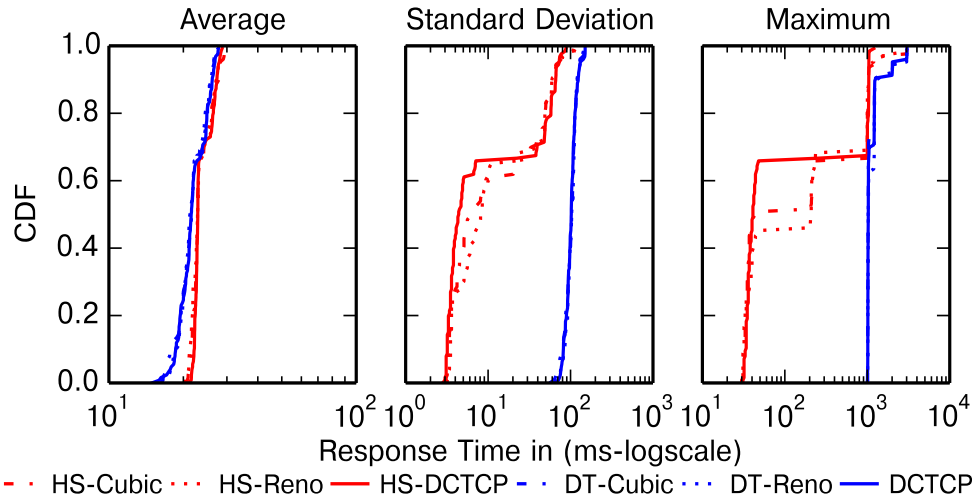
Figure 12: *Testbed topology for HSCC evaluation*

In this set of experiments, we deploy the NetFPGA-based IncastGuard switch in a small-scale real-testbed with same configuration and setup of FairSwitch and IncastGuard deployments in Section **??** and Section **??**, respectively. Figure 12 shows the topology used in the experiments and Figure 12b shows a photo taken of the real-life deployment. The objectives of the following micro-benchmark experiments are: *i)* to verify that with the support of HSCC, TCP can support many more connections and maintains high link utilization; *ii)* to verify effectiveness of HSCC system in reducing incast congestion effect on TCP flows; *iii)* to quantify HSCC's ability to improve the FCT of mice when competing for the bottleneck link with elephants.

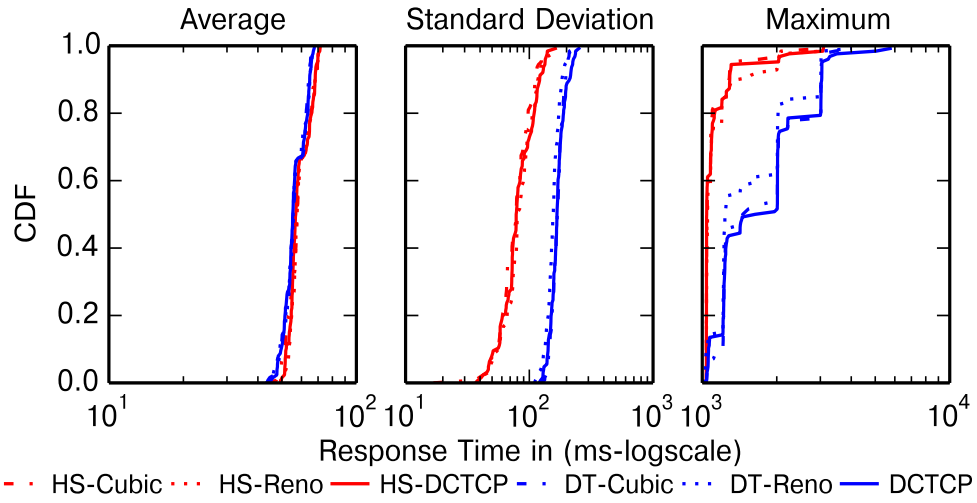
Incast Traffic without Background Workload: First, we run two mild and heavy incast scenarios where a large number of mice flows transfer 11.5KB sized blocks.

Experimental Setup: in both scenarios, 7 servers in rack 4, issue 100 web requests to retrieve **"index.html"** webpage of size 11.5KB from the other 21 servers in rack 1, 2 and 3. In this scenario, each requester end-host uses 2 parallel TCP connections to satisfy the 1000 requests. Hence, a total of 252 $((21 \times 7 - 21) * 2)$ synchronized requests are issued. In the heavy load case, a thousand consecutive requests are issued however, each process uses 5 parallel TCP connections instead of 2. This results into 630 flows (i.e., 126×5) at the same time.

Experimental Results: Figure 13 shows, under both mild and heavy load, HSCC achieves a significantly improved performance for TCP flows. The com-



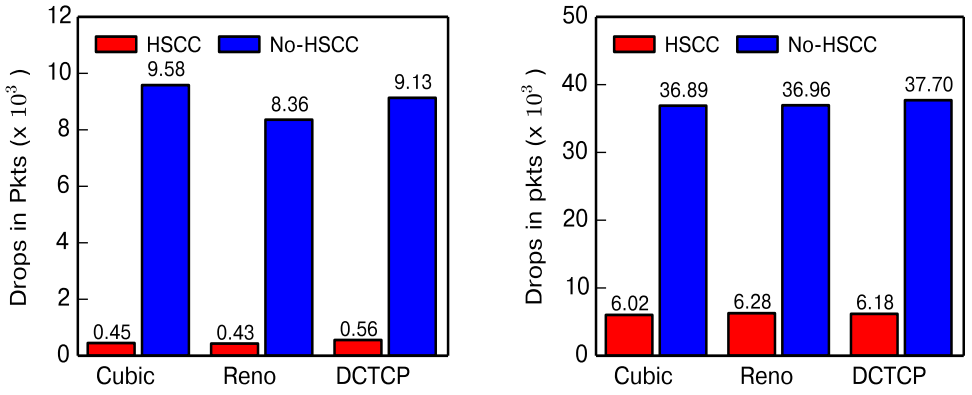
(a) *Medium case: 126 mice (incast) flows*



(b) *Heavy case: total of 630 mice (incast) flows*

Figure 13: *Experimental results of two Incast moderate and heavy scenarios. The metrics are average, standard deviation and maximum FCT for Cubic, Reno or DCTCP w/wo HSCC. Each flow is 1000 11.5KB blocks.*

peting 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 and New-Reno (henceforth abbreviated as Reno) with



(a) *Medium case* **(b)** *Heavy case*

Figure 14: *Experimental results of two Incast moderate and heavy scenarios. The figures show Total Packet Drops for Cubic, Reno or DCTCP w/wo HSCC.*

DropTail and DCTCP. In addition, it can improve the FCT of the end of the tail (maximum FCT) by two orders-of-magnitude suggesting that almost all flows (including tail-ends) can meet their deadlines. In the heavy case, it can also achieve noticeable improvements even with 630 flows competing all together. each with 1 MSS (1460 bytes) of window the total traffic of 920KB is $\approx 3.2\%$ larger than the size of the bottleneck pipe or, 287KB which is the switch Buffer size plus the bandwidth delay product. Finally, HSCC can efficiently detect the incast and proactively throttle the flows to avoid packet drops, Figure 14 shows that, it can significantly decrease the drop rate during incast events by $\approx 96\%$ in medium load compared to only $\approx 86\%$ in heavy load scenarios.

Mild 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.

Experimental Setup: incast flows is set to compete with elephants flows for the same outgoing queue. To this end, 21 iperf [15] long-lived flows is 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 1.15 MB file partitioned into 100 11.5KB chunks totaling $\approx$ 145 MBytes) after elephants have reached steady state (i.e., at the 10^{th}sec).

Experimental Results: Figure shows that, in medium load, HSCC achieves a FCT improvements for mice while nearly not affecting elephants' performance. Mice flows benefit with HSCC by improving on FCT on average and again with one order-of-magnitude reduction in FCT standard deviation compared to TCP (Cubic, Reno) with DropTail and DCTCP. Also, in terms of the tail (i.e., the last and similarly the 99%), HSCC reduces the tail FCT by two order-of-magnitude almost close to the average and within 10's of ms. The improvement means mice flows finish quickly within their deadlines. Figure 15b shows that elephant flows are almost not affected by HSCC's intervention by throttling their rates during the short incast periods. In Figure 15c, the drops under HSCC is reduced by its efficient rate control during incast hence mice flows avoid long timeouts of at least 200ms.

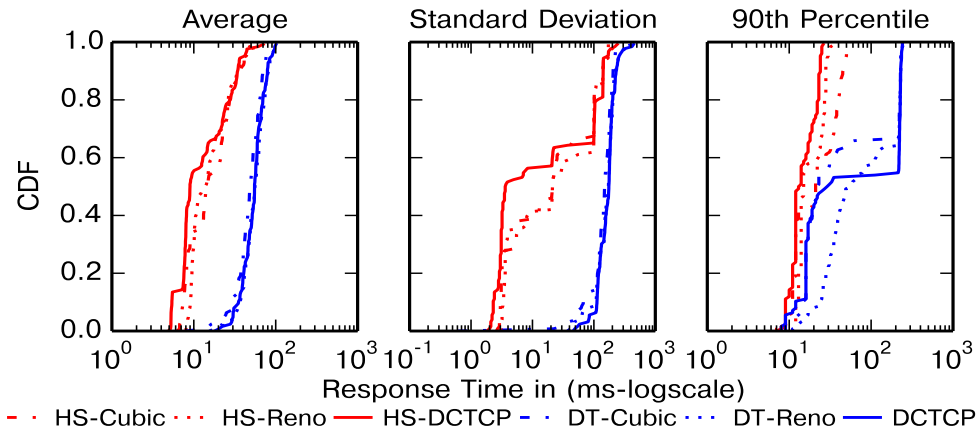
Heavy Incast Traffic with Background Workload: We repeat the above experiment, increasing the frequency of Mice incast epochs to 9 times within the 20 second period (i.e., at the 2^{nd}, 4^{th}, .., and 18^{th} sec).

Experimental Setup: in each epoch, each server requests a 1.15MB file partitioned into 100 11.5KB chunks totaling $\approx$ 145 MBytes per epoch and a total of $\approx$ 1.3 GBytes for all 9 epochs.

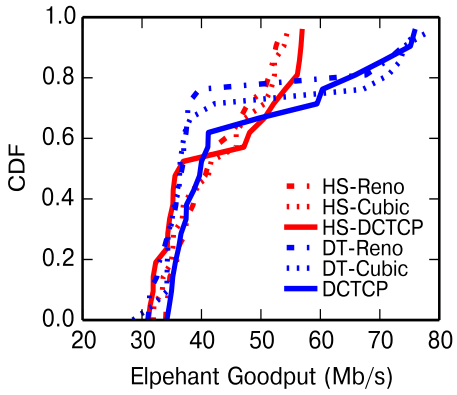
Experimental Results: as shown in Figure 16a, even with the increased incast frequency, HSCC scales well with higher incast rates even-though mice flows are also fighting their way against fat elephant flows. Mice flows' average and standard deviation of FCT see similar improvement as the previous experiments 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 chances of experiencing timeouts as show in Figure16c. Compared to the previous experiment, Figure 16b 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.

6.2 Datacenter Workloads based Experimental Results

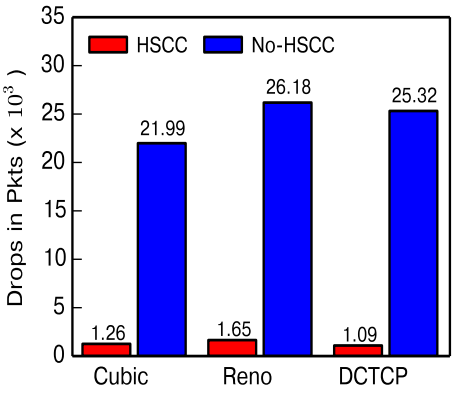
We use the traffic generator described in Section 2, to run the experiments involving websearch and datamining traffic workloads. In addition, we resuse iperf program [15] to emulate long-lived background traffic (e.g., VM migrations, backups and so on) in certain scenarios. We setup a scenario to create an One-to-All experiment w/wo background traffic. In One-to-All, clients running on the VMs in one rack send requests randomly to any of all other servers in the cluster. If back-



(a) *The average, standard deviation and 90th percentile of mice flows*



(b) *AVG elephant throughput*

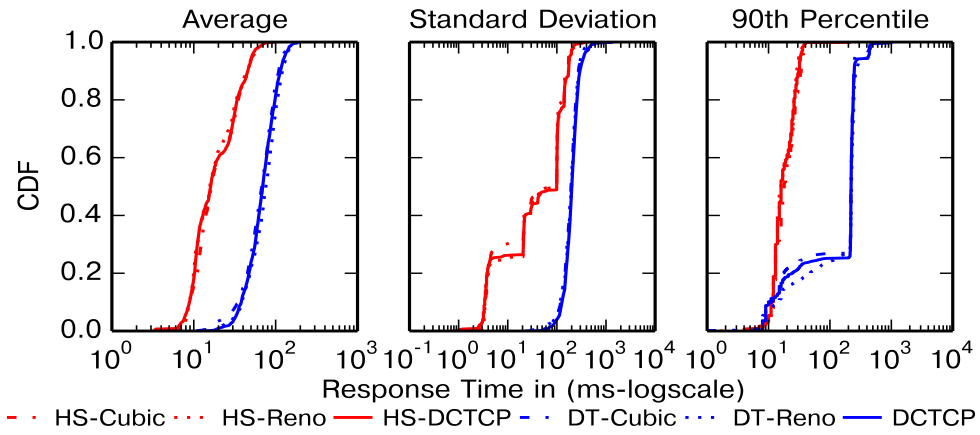


(c) *Total packet drops*

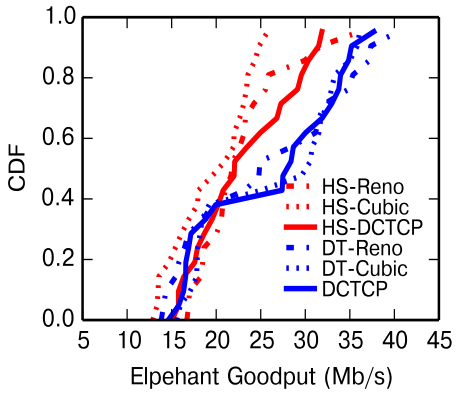
Figure 15: *254 mice flows with 21 elephants: performance of HSCC vs (Cubic or Reno TCP with DropTail or DCTCP) is reported. Each of the 256 mice flow requests 1.15MB file (= 100 × 11.5KB) 1 time while competing with 21 elephants*

ground traffic is introduced, we run a long-lived iperf flows in all-to-all fashion to mimic sudden and persistent spike in network load. Even though, we classify flows with size $<=$ $100KB$ as small, $>$ $100KB$ *and* $<=$ $10MB$ as medium and $>=$ $10MB$ as large. In the following experiments, We focus more on the small flows which are the target for HSCC.

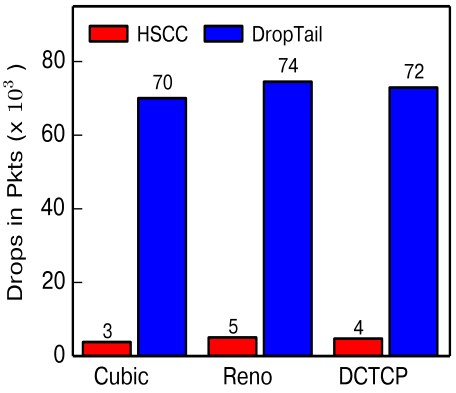
A Scenario without Background Traffic: we run One-to-All scenario and report the performance of average, median and maximum FCT and number of



(a) *The average, standard deviation and 90th percentile of mice flows*



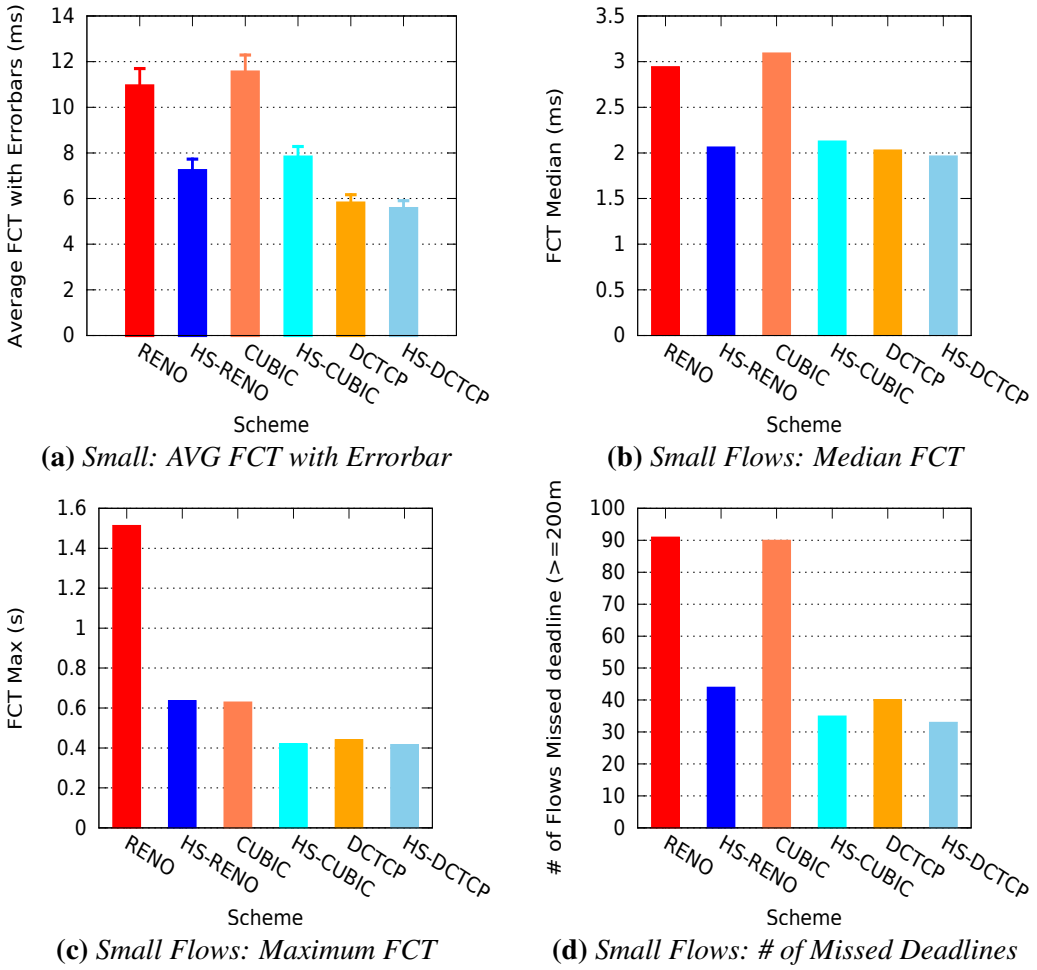
(b) *AVG elephant throughput*



(c) *Total packet drops*

Figure 16: *630 mice flows with 21 elephants: performance of HSCC vs (Cubic or Reno TCP with DropTail or DCTCP) is reported. Each of the 630 mice flow requests 1.15MB file (= 100 × 11.5KB) 1 time while competing with 21 elephants*

flows who missed a 200ms deadline for in the small flows. The traffic generator is set to randomly initiate 1000 requests per server per rack to randomly picked servers on one of the other racks. Figures 17a, 17b, 17c and 17d show the average, median and max FCT and missed deadlines for small flows in websearch workload. While, Figures 18a, 18b,18c and 18d show the average, median and max FCT and missed deadlines for small flows in datamining workload. We make the following observations: *i)* For websearch workloads, HSCC improves per-



(a) *Small: AVG FCT with Errorbar*

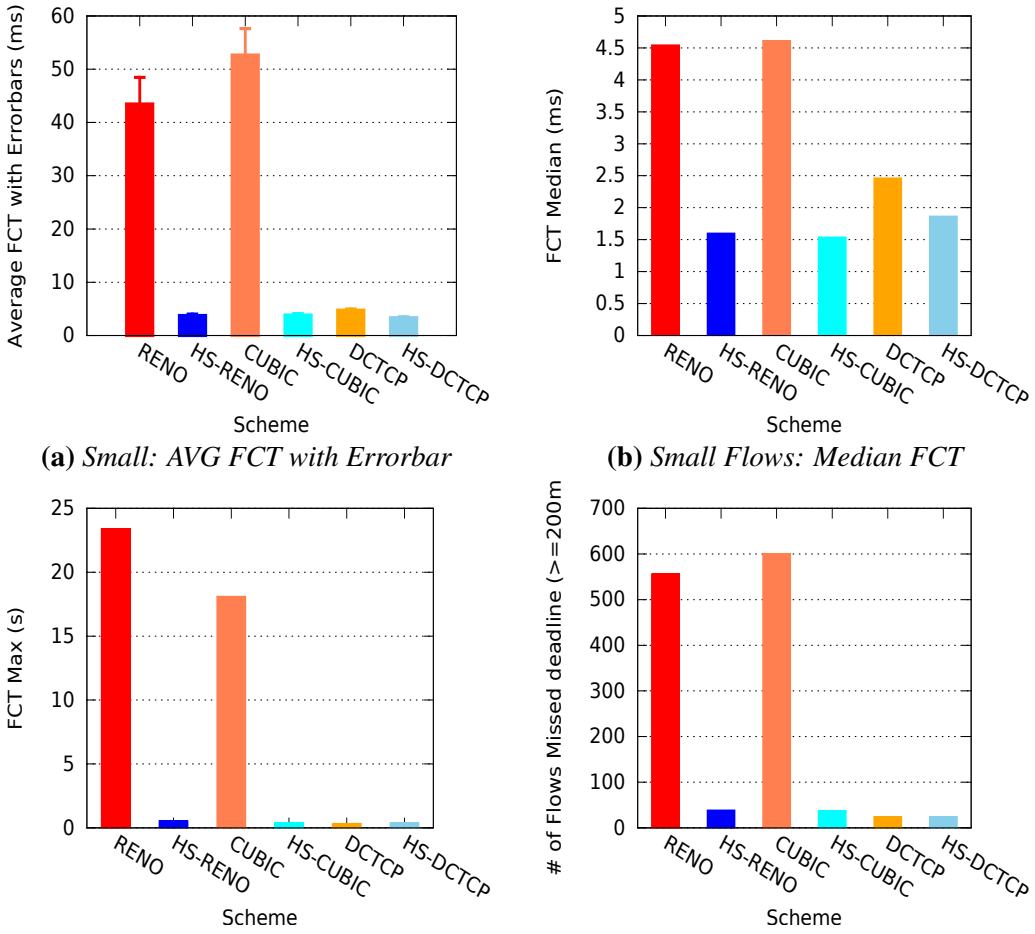
(b) *Small Flows: Median FCT*

(c) *Small Flows: Maximum FCT*

(d) *Small Flows: # of Missed Deadlines*

Figure 17: *Performance metrics of One-to-All Websearch workload with no background traffic*

formance of small flows of various TCP variants in both the average, maximum and missed deadlines FCTs. 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 workload, the improvements are even more significant due to domination of small flows population in this workload. For instance, compared to Reno, Cubic and DCTCP, HSCC re-



(a) *Small: AVG FCT with Errorbar*

(b) *Small Flows: Median FCT*

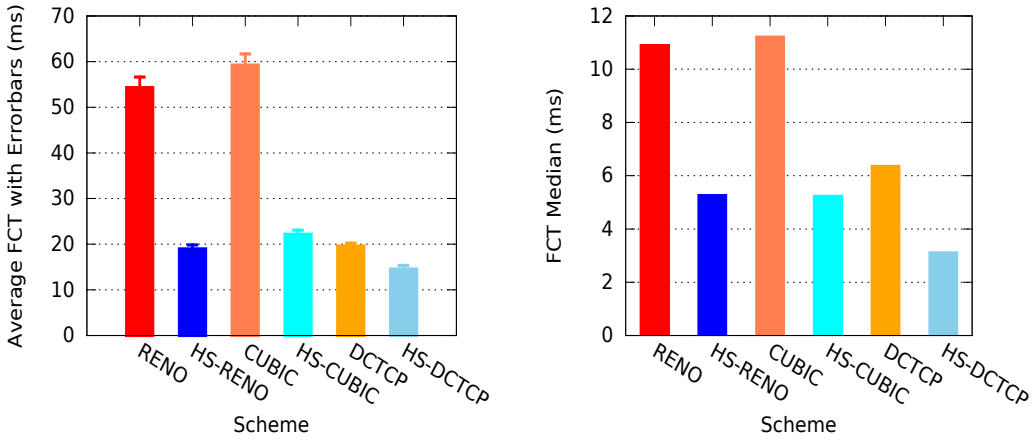
(c) *Small Flows: Maximum FCT*

(d) *Small: # of Missed Deadlines*

Figure 18: *Performance metrics of One-to-All Datamining workload with no background traffic*

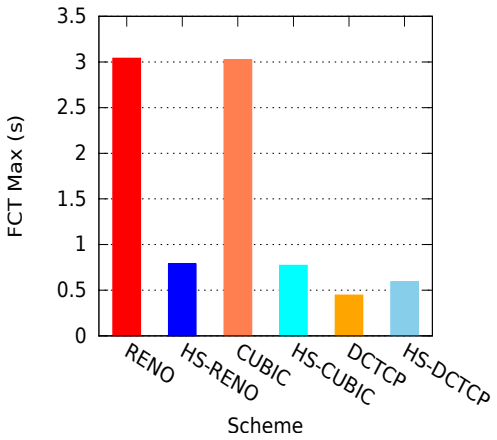
duces the FCT of small flows by $\approx (92\%, 93\%, 30\%)$, $\approx (65\%, 67\%, 25\%)$ and $\approx (98\%, 98\%, -\%)$ on the average, medium and maximum, respectively. In addition, the number of missed deadlines is improved by $\approx (93\%, 94\%, 4\%)$ for Reno, Cubic and DCTCP, respectively. *iii)* We notice that DCTCP improves FCT over its RENO and CUBIC counterparts and HSCC could improve DCTCP performance in websearch and datamining workloads.

A Scenario with Background Traffic: to put HSCC under a true stress, we run the same One-to-All scenario along with an all-to-all long-lived background

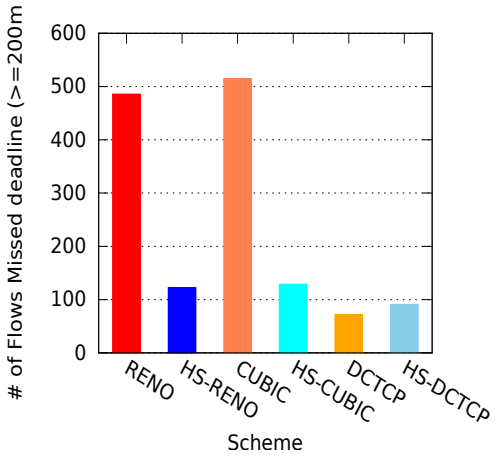


(a) *Small: AVG FCT with Errorbar*

(b) *Small Flows: Median FCT*



(c) *Small Flows: Max FCT*



(d) *Small: Missed Deadlines*

Figure 19: *Performance metrics of One-to-All websearch workload with background traffic.*

traffic during the experiment. We report similar metrics as in the aforementioned case. Figures 19a, Figure 19b, 19c and 19d show the average, median and max FCT and missed deadlines for small flows in this scenario. We observe the following: *i)* HSCC improves further the average, median and maximum FCT of small flows regardless of TCP congestion control in use. As shown in the results, compared to Reno, Cubic and DCTCP, HSCC reduces the FCT of small flows by $\approx (66\%, 63\%, 26\%)$, $\approx (52\%, 54\%, 51\%)$ and $\approx (75\%, 75\%, -)$ on the average and median, respectively. HSCC managed to decrease the maximum FCT

and consequently number of missed deadlines by $\approx (75\%, 75\%)$ for Reno, Cubic, respectively. However, DCTCP sees slight increase in the max FCT and missed deadlines, which might be attributed for its reaction to excessive ECN marks caused by background flows.

In summary the experimental results support and highlight the performance gains (esp. for time-sensitive applications) obtained by adopting HSCC system. In particular, they show that:

- HSCC minimizes the mean and variance of mice flow completion times and significantly reduce by 1-2 orders-of-magnitude the FCT for the tail end.
- HSCC can improve further if the bandwidth-hungry elephants are hogging the network.
- HSCC efficiently handles mice traffic, even in low and high frequency incast.
- HSCC achieves its goals with no more than default assumptions about the network stack and without any modifications to guest VMs.

7 Related Work

Much work has been devoted to addressing congestion problems in datacenters and in particular incast congestion. Recent works [9, 33, 8] 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 datacenter ill-suited timeout mechanism and use of large initial congestion windows in TCP's congestion control.

Towards solving the incast problem, one of the first works [20] 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 [29] suggested modifying the TCP protocol in data centers by reducing the value of the minRTO value from 200ms to microseconds scale. Then DCTCP [1] and ICTCP [31] were proposed as a 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 [34] 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 [25] is another congestion control mechanism for datacenters 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.

8 Report Summary

In this report, we showed empirically that the low bandwidth-delay product of datacenters results into excessive RTOs and the short TCP loss cycle was found to be partially blamed for it. We have demonstrated analytically that short cycles can greatly degrade TCP performance when the losses at the end of the cycle are only recover-able via RTO. To improve the performance of short TCP flows, we have proposed stretching the period of TCP cycles in datacenters. To this end, we designed an efficient control theory inspired hysteretic switching mechanism namely HSCC. The proposed system improves the FCT of most TCP small flows who are shown to account for the large part of flows generated by data center workloads. The design is based on the outcomes of our empirical and behavioral analysis of TCP in datacenters. HSCC consists of a switch implementing a two-threshold hysteresis control laws to proactively switch between TCP mode and a slower CBR mode. The simulations and testbed experiments show that HSCC improves the FCT for mice traffic without impacting the progress of elephant flows. Similar to RWNDQ and IQM, HSCC stands for our principle point in this thesis which is improving TCP performance in public cloud networks where modifications to TCP stack of guest VMs are prohibited. We believe further experiments of our switch-based schemes are necessary in large data centers and we will also negotiate with switch manufacturers possible adoption of our switch-based schemes in their ASIC prototypes. However, from our experience, most switch-based schemes see slow real adoption and they have to go through rigorous and length testing and production cycles. For this reason, in the next part, we

explore hypervisor-based approach to solve similar TCP flow problems observed in data centers. Hypervisor-based approaches are more appealing and may see immediate adoption because they require no modification to the guest VMs nor the switching devices.

References

[1] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data center TCP (DCTCP). *ACM SIGCOMM Computer Communication Review*, 40:63, 2010.

[2] M. Alizadeh, A. Kabbani, B. Atikoglu, and B. Prabhakar. Stability analysis of QCN. *ACM SIGMETRICS Performance Evaluation Review*, 39(1):49, 2011.

[3] M. Alizadeh, S. Yang, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker. Deconstructing datacenter packet transport. In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks (HotNets)*, pages 133–138, 2012.

[4] W. Bai, 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 Conference on Networked Systems Design and Implementation (NSDI)*, NSDI'15, pages 455–468, Berkeley, CA, USA, 2015. USENIX Association.

[5] W. Bai, L. Chen, K. Chen, and H. Wu. Enabling ECN in multi-service multi-queue data centers. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation (NSDI)*, pages 537–549. USENIX Association, 2016.

[6] T. Benson, A. Akella, and D. A. Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement (IMC)*, IMC '10, pages 267–280, New York, NY, USA, 2010. ACM.

[7] C. Casetti, M. Gerla, and S. Mascolo. TCP Westwood : End-to-End Congestion Control for Wired/Wireless Networks. *Wireless Networks*, 8:467–479, 2002.

[8] W. Chen, F. Ren, J. Xie, C. Lin, K. Yin, and F. Baker. Comprehensive understanding of TCP Incast problem. In *Proceedings of the IEEE INFOCOM Conference*, 2015.

[9] Y. Chen, R. Griffith, J. Liu, R. H. Katz, and A. D. Joseph. Understanding TCP incast throughput collapse in datacenter networks. In *Proceedings*

of the 1st ACM workshop on Research on enterprise networking (WREN), pages 73–82, 2009.

[10] 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 *Proceedings of the 11th USENIX Symposium on Networked Systems (NSDI) Design and Implementation (NSDI 14)*, pages 17–28, 2014.

[11] S. Fahmy and T. P. Karwa. Tcp congestion control: Overview and survey of ongoing research. Technical report, Purdue University, 2001.

[12] 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 the ACM SIGCOMM*, volume 09, pages 51–62, 2009.

[13] S. Ha, I. Rhee, and L. Xu. CUBIC : A New TCP-Friendly High-Speed TCP Variant. *ACM SIGOPS Operating Systems Review*, 42(5):64–74, jul 2008.

[14] C. Hollot, V. Misra, D. Towsley, and Weibo Gong. Analysis and design of controllers for AQM routers supporting TCP flows. *IEEE Transactions on Automatic Control*, 47(6):945–959, jun 2002.

[15] iperf. The TCP/UDP Bandwidth Measurement Tool. https://iperf.fr/.

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

[17] M. H. Jim Keniston, Prasanna S Panchamukhi. Kernel probes (kprobe). https://www.kernel.org/doc/Documentation/kprobes.txt.

[18] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken. The nature of data center traffic. In *Proceedings of the ACM SIGCOMM conference on Internet measurement (IMC)*, New York, New York, USA, 2009.

[19] H. K. Khalil. *Nonlinear systems*. Prentice Hall, 2002.

[20] 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 '07 - PDSW '07*, 2007.

[21] M. Kuhlewind, D. P. Wagner, J. M. R. Espinosa, and B. Briscoe. Using data center TCP (DCTCP) in the Internet. In *Proceedings of the IEEE Globecom Workshops*, 2014.

[22] M. Mathis, J. Semke, J. Mahdavi, and T. Ott. The macroscopic behavior of the TCP congestion avoidance algorithm. *ACM SIGCOMM Computer Communication Review*, 1997.

[23] M. Mellia, I. Stoica, and H. Zhang. TCP Model for Short Lived Flows. *IEEE COMMUNICATIONS LETTERS*, 6(2), 2002.

[24] V. Misra, W.-B. Gong, D. Towsley, 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 *Proceedings of the ACM SIGCOMM*, volume 30, pages 151–160, New York, New York, USA, 2000. ACM Press.

[25] 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 *Proceedings of the ACM SIGCOMM*, SIGCOMM '15, pages 537–550, New York, NY, USA, 2015. ACM.

[26] A. Shieh, S. Kandula, A. Greenberg, C. Kim, and B. Saha. Sharing the data center network. In *Proceedings of the 8th USENIX NSDI Conference*, 2011.

[27] B. Stephens, A. L. Cox, A. Singla, J. Carter, C. Dixon, and W. Felter. Practical DCB for improved data center networks. In *Proceedings of the IEEE INFOCOM Conference*, 2014.

[28] K. Tan, J. Song, Q. Zhang, and M. Sridharan. A Compound TCP Approach for High-Speed and Long Distance Networks. In *Proceedings of the IEEE INFOCOM Conference*, pages 1–12. IEEE, 2006.

[29] 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*, 39:303, 2009.

[30] D. X. Wei, C. Jin, S. H. Low, and S. Hegde. FAST TCP: Motivation, Architecture, Algorithms, Performance. *IEEE/ACM Transactions on Networking*, 14(6):1246–1259, dec 2006.

[31] H. Wu, Z. Feng, C. Guo, and Y. Zhang. ICTCP: Incast congestion control for TCP in data-center networks. *IEEE/ACM Transactions on Networking*, 21:345–358, 2013.

[32] E. Zahavi, A. Shpiner, O. Rottenstreich, A. Kolodny, and I. Keslassy. Links as a Service (LaaS). In *Proceedings of the ACM/IEEE Symposium on Architectures for Networking and Communications (ANCS)*, 2016.

[33] Y. Zhang and N. Ansari. On mitigating TCP Incast in Data Center Networks. In *Proceedings of the IEEE INFOCOM Conference*, pages 51–55, 2011.

[34] 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.