Noname manuscript No.
(will be inserted by the editor)

Mitigating Incast-TCP Congestion in Data Centers with SDN

Ahmed M. Abdelmoniem · Brahim Bensaou · Amuda James Abu

the date of receipt and acceptance should be inserted later

Abstract In data center networks (DCNs), in the presence of long lived flows that tend to bloat the switch buffers, short-lived TCP-incast traffic suffers repeated losses that are often recovered via timeout. The minimum retransmission timeout (minRTO) in most TCP implementations being fixed to around 200ms, a whopping three orders of magnitude the actual RTT in DCNs, interactive applications that often generate incast traffic tend to suffer unnecessarily long delays. The best and most direct solution to such problem would be to customize the minRTO to match DCNs delays, however, this is not always possible; in particular in public data centers where multiple tenants, with various versions of TCP, co-exist. In this paper, we propose the next best thing, by using techniques and technologies that are already available in most commodity switches and data centers and that do not interfere with the tenant's virtual machines/TCP protocol. We invoke the programmability of SDN switches and design an SDN-based Incast Congestion Control (SICC) framework that uses a SDN network application and a shim-layer at the host hypervisor to mitigate incast congestion. We demonstrate the performance of the proposed scheme via real deployment in a small-scale testbed and ns2 simulation in larger environments.

Ahmed M. Abdelmoniem* E-mail: amas@cse.ust.hk
Brahim Bensaou E-mail: brahim@cse.ust.hk
Amuda James Abu E-mail: ajabu@cse.ust.hk
Department of Computer Science and Engineering
The Hong Kong University of Science and Technology
Clear Water Bay, Hong Kong
*Contact corresponding Author for inquiries and feedback.
The source code, simulation and experiments are available at `https://ahmedcs.github.io/SICC/`
This paper is an extended version of [4].

Keywords Congestion Control · Data Center Networks · Incast · Software Defined Networking · TCP.

1 Introduction

Driven by the popularity of cloud computing, public data center network (DCNs) abound today in applications that generate a large number of traffic flows with varying characteristics and requirements. These range from large groups of barrier-synchronized, short-lived, time-sensitive flows, like those resulting from web searches and all partition/aggregate applications that result in the so-called TCP-incast traffic (called in the remainder mice); to long-lived, time-insensitive, flows that benefit gratly from high throughput, such as those resulting from backups and virtual machine migration (referred in the sequel as elephants). In particular, recent studies [8, 11, 23] have shown that while in practice the lion's share in terms of traffic volume still goes to the elephants, DCNs are highly crowded with mice flows.

DCNs are structured to provide a high bandwidth with low latency, as such the traffic inertia[1] in such networks is small, thus, the amount of buffer space required to absorb to absorb such traffic bursts is also small. As a result, DCNs mostly use Ethernet switches with small buffers (instead of routers with large buffers) to interconnect the servers. However, in the presence of such small buffers, the sudden surge of synchronized new incast-TCP traffic still results in congestion events, in particular due to the presence of elephant traffic that

[1] We define the traffic inertia as the amount of traffic that can be injected in the network by a flow before congestion takes place. this is typically the fair share of so-called bandwidth-delay product for that flow

(by design of TCP) bloat the shared buffer. Such complex congestion events have been shown in recent works [6, 23] to be inadequately handled by traditional TCP, as TCP is agnostic to the latency requirements of mice traffic flows as well as to the composite nature of the application data.

Recent works [1, 3, 6, 12, 21, 23] addressed this issue via different approaches. Many such works adopts an -end-to-end TCP-AQM as a means to maintaining a small queue in the switches, ensuring thus a high throughput for elephants while maintaining empty buffer space for burst arrivals. For instance, DCTCP [6] modifies TCP reaction to ECN marks to cut the congestion windon (*cwnd*) in proportion to the number of marks received per RTT. While the marking in the switch is done based on the instantaneous buffer occupancy. Identifying the difficulty of imposing a standard TCP on the VMs in public data centers, in our earlier work, RWNDQ [3] we proposed a switch-based fair allocation scheme that does not need to change TCP. Instead, in our scheme, the switch calculates a flow fair share and modifies the advertised receiver window (*rwnd*) when congestion is building up, to impose this fair share on all the the sources regardless of their nature (mice or elephants). Both approaches involve modification of either the TCP stack in the guest VMs or the switch software which are non-appealing for immediate deployment in large DC networks.

Software Defined Networking (SDN) [15] was recently adopted as an emerging network routers and switches design approach that separates the control functions from the datapath, relinquishing the control function to a dedicated central controller(s) with a global-view of the network. OpenFlow [13] is currently the dominant standard interface between the controller and the data path. This technique enables rich network control functions (such as routing, security, admission control and so on) to be easily implemented and deployed on top of the network operating system in the SDN controller as simple applications. To avoid modifications to the TCP stack or the switch, In this paper, we invoke the programmability of SDN switches to design an SDN-based incast-TCP congestion control (SICC) framework that uses a SDN network application in the controller and a shim-layer at the host hypervisor to mitigate incast congestion.

1.1 Motivation and Objectives

A good solution to the incast congestion problem should be appealing to both the tenant and the cloud operator. Hence, we argue that modifying the TCP protocol and/or the hardware switching logic can only apply to small scale private data centers. In particular, in most public cloud services, tenants can upload their own operating system images to their virtual machines and modify/fine tune their protocol stack as needed. In addition, while modifying the switch hardware is feasible from the cloud provider perspective, it remains an unappealing costly perspective. As a result, our target in this paper is to design a solution to the incast problem that has the following requirements: (R1) Effectiveness: to demonstrate effective handling of the problem of incast traffic congestion by significantly improving the incast flow completion time, without degrading the throughput of elephant flows; (R2) Scalability: to avoid modifying the TCP sender/receiver protocol, nor alter the hardware switches. (R3) Simplicity and Practicality: to enable immediate deployment potential (via simple software patches) by restricting any changes to software in the programmable devices (e.g., switch controllers and/or hypervisors/vswitches) that are fully under the control of the DCN operator; .

To achieve these objectives, we adopt a triangular approach where the OpenFlow enabled switches report their statistics to the controller, the controller estimates the extent of congestion and notifies the hypervisors to quench the TCP sources that contribute to this congestion. This can be done programmatically, by implementing a SDN control application and a shim layer in the hypervisor.

1.2 Summary of Contributions

1. We explore the prospect of using SDN paradigm to design congestion control schemes suited for data center networks.
2. We develop an easily deployable SDN framework that handles incast congestion event via cooperation among the SDN controller and the end-host hypervisor.
3. We implement the proposed framework and evaluate its performance via simulation and real-testbed experiments.

2 Related Work

Recent years have seen an increasing activity in the design of congestion control mechanisms for DCNs. In general, these works fall into one of four categories:

1. **Sender-based protocols**: in [21], it is observed that there is a mismatch between the standard TCP retransmission timer in the hosts and the actual round-trip times (RTTs) experienced in DCNs. Modifying the sender TCP stack to use high-resolution

timers was thus proposed to enable TCP timeout detection in the microsecond granularity level. The so-called DCTCP [6] proposed modifying TCP congestion window adjustment function to react proportionally to the congestion level. RED-AQM parameters are tuned to enforce a small ECN-marking threshold to achieve a small queue length. Both approaches can achieve small delays for mice traffic but require modifications of the TCP sender and receiver algorithms as well as fine tuning of RED parameters at the switches for DCTCP.

2. **Receiver-based protocols**: ICTCP [23] was proposed as a modification to TCP receiver to handle incast traffic. ICTCP adjusts the TCP receiver window proactively, before packets are dropped. The experiments with ICTCP in a real testbed show that ICTCP can almost curb timeouts and achieves a high throughput for TCP incast traffic. Unfortunately, ICTCP does not address the impact of buffer bloating issues caused by the co-existence of elephants in the same buffer as mice. Furthermore, it is effective only if the incast congestion happens at the destination node and finally it also requires changes to the TCP receiver algorithm.
3. **Switch-assisted protocols**: in [1–3] we proposed AQM schemes to regulate TCP sending rate with minor modifications to the DropTail AQM. RWNDQ [1, 3] tracks the number of established flows to calculate a fair share for each flow and updates the TCP receiver window in the ACK to feedback this explicit share to TCP sources. IQM [2] in contrast monitors the TCP connection setup and tear-down events at the switch to forecast the imminence of possible incast congestion and resets the receiver window of the reverse path ACKs to 1 MSS to slow down elephants, in the coming few RTTs, making thus room for the forthcoming incast traffic. Both schemes are shown to curb timeouts for incast traffic and achieve a high throughput for elephant traffic, however, both require switch software modifications.
4. **SDN-based**: SDTCP [12] involves the SDN controller in monitoring in-network congestion messages triggered by OpenFlow switches and select currently active elephant flows. The controller sets up OpenFlow rules at the switches to decrease the sending rate of elephants via rewriting the TCP receive window of ACKs. The experiments conducted in an emulation environment (Mininet) shows almost zero loss for TCP incast without major effect on the goodput of the elephants. However, the proposed modifications and congestion notification messages from the switches are unrealistic unless they are implemented by modifying the switches.

3 Proposed Methodology

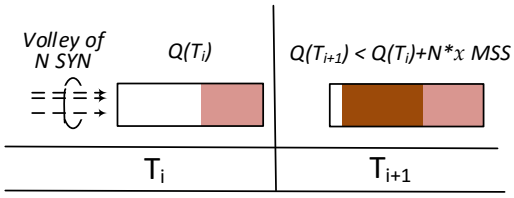


Fig. 1: SICC Idea Rationale

The basic idea behind our proposed SICC framework is illustrated in Fig. 1. Considering the buffer occupancy at a switch outgoing port, assuming that mice flows are short-lived, they will mainly contribute to the queue variation in the port buffer, in contrast, the persistent queue length is mainly due to the contribution of elephant flows in bloating the buffer. As a result, denoting $Q(T_i)$ the persistent queue at round i of duration T_i (e.g., RTT i), the arrival of a volley of N new TCP connections (as indicated by the arrival of N TCP SYN packets) would lead the queue at period T_{i+1} to be no more than $Q(T_i)+N*x*MSS$ bytes, x being the initial window size of TCP. To control congestion, one may simply inhibit the congestion window and deploy flow control whenever $Q(T_{i+1})$ is expected to exceed the buffer size. For instance in such situation each flow is allowed to send only 1 MSS per RTT. To ensure a high link utilization, flow control is turned off whenever the queue reaches a given low threshold, allowing thus the sources to recover the use of their already established congestion window. The underlying principle with this approach is to achieves fairness in the short terms among ongoing flows (mice and elephants) whenever a salvo of mice flows is starting. As mice are expected to be short-lived, knowing that the persistent queue is mainly due to elephants, flow throttling is deactivated once the queue drops below the threshold, hence meeting requirement (R1) above.

In principle, regardless of the TCP-flavour, the TCP source sending rate is determined by the sender window $swnd = \min(rwnd, cwnd)$, with $rwnd$ and $cwnd$, being the advertised receiver window and the current congestion window respectively. Since $cwnd$ is normally at least equal to 2 MSS, setting $rwnd$ in incoming TCP ACKs to 1 MSS during incast congestion will have the immediate effect of throttling all the ongoing flows. The direct effect of this is to ensure short term fairness among all the flows during the incast period. This can be easily implemented as a switch-based algorithm to avoid modifying the TCP source/receiver, however

the cost of changing all the switches is prohibitive. Instead, and to meet requirement (R2), the SDN controller, being aware of flow arrivals and thus the possible incast events, controls when the end host hypervisor is to rewrite *rwnd* field in the incoming TCP ACK headers. SDN also provides much useful statistics on the ongoing number of flows and the queue occupancy for each switch port. Noticing that all the rewriting happens in the hypervisor below the virtual machines and that *rwnd* processing is universal to all TCP implementations, our framework is obviously transparent to the TCP variant deployed inside the VM. In addition during the incast period when our controller is in action all flows regardless of the TCP-flavor in use receive the same share of bandwidth.

To meet requirement (R3), i.e., simplicity, instead of tracking individual flow states to estimate accurately the queue length in the next interval, the controller uses rough estimates by simply counting the accumulated number N of TCP segments with a SYN-ACK bit, reduced by the number of TCP segments with the FIN bit set; in the worst case, this rough estimate results in a conservative estimate of the predicted queue length. (Without loss of generality, in the sequel we will consider the value of initial TCP congestion window (x) to be 1 MSS.)

Figure 2a shows the detailed protocol interactions among the different modules residing on the controller, switches and end-hosts as follows: 1) The controller uses a monitoring module/application to track and extract information (e.g., the window scaling option) from incoming SYN/FIN. This is done by setting SYN-copy rules in all ToR switches in the data center. 2) The controller uses a warning module/application to predict imminent incast congestion events based on SYN/FIN arrival rates and the current queue length. In case of possible congestion, incast ON/OFF special messages are directed to the involved senders' VM addresses. 3) In the hypervisor/vswitch SICC monitoring module, upon receipt of incast ON message for a certain VM, all incoming ACK packets for that VM are intercepted and their *cwnd* rewritten, until an incast OFF message is received later or the average duration of typical mice (short-livd) flows is exceeded. 4) SDN switches only need to be programmed with a Copy-to-Controller rule for SYN/FIN packets, the controller will set out a rule at the DC SDN-switches to forward a copy of any SYN/FIN packet through the south-bound API (OpenFlow) protocol interface.

Figure 2b illustrates a possible deployment scenario of our proposed SICC framework in a SDN based data centers.

4 SDN-based Incast Congestion Control

The main variables and parameters used in the SICC framework are described in Table 1. Notice that T, DM, α_1 and α_2 are system parameters as described in Table 1.

Table 1: Variables and Parameters used in SICC framework by the SDN network application and the hypervisor shim-layer

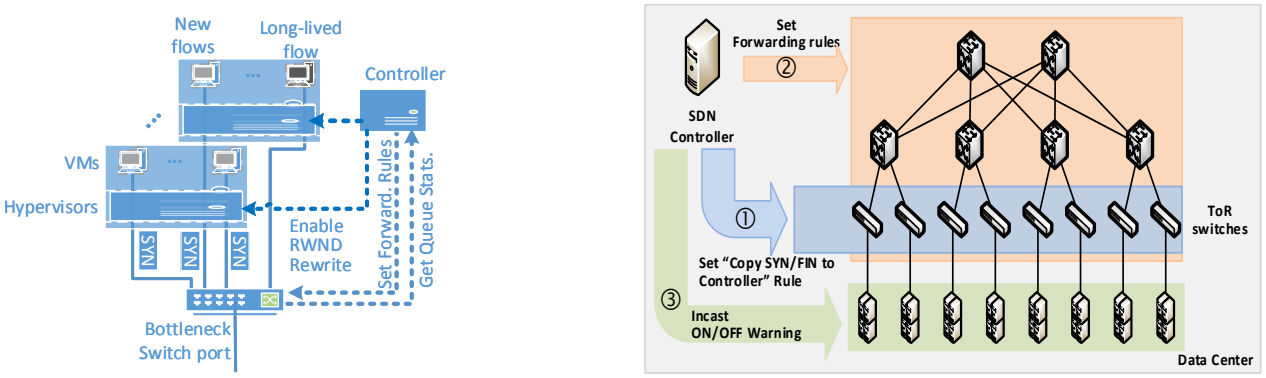
Parameter name	Description
T	Timeout value for monitoring interval
α_1	Queue threshold to turn OFF Incast
α_2	Queue threshold to turn ON Incast
DM	Average runtime (duration) for mice flows to finish
List Objects	Description
$SWITCH$	List of the controlled SDN switches
$SWITCH\_PORT$	List of the ports on the switches
$PORT\_DST$	List of destinations reachable though port
DST_SRC	List of destinations and source pairs
Q	Average length of the output queue q
B	buffer size on the forward path
W	Window scale of source-destination pair
M	Maximum segment size of source-destination pair
β	Coarsely estimated differential of new connections
κ	Boolean true if incast is ON

4.1 SICC: Incast detection network application

The network application communicates with the central controller via the north-bound API in order to set OpenFlow rules at OpenFlow-based switches in the data center. These rules instruct the switches to forward a copy of any *SYN* or *FIN* packets to the SICC application for further processing. In most cases, TCP *SYN* packets contain optional TCP header fields with useful information (i.e., maximum segment size and window scaling value). Such information is stored in source-destination-based hash tables to be used in by the SICC application. Typically, the controller probes regularly for switch port statistics over a fixed interval which SICC uses to calculate a smooth weighted moving average of the queue occupancy. Hence, the SICC application can predict possible congestion events using the following algorithm:

The SICC network application shown in Algorithm 1 is an event-driven mechanism that implements two major event handlers: packet arrivals and incast detection timer expiry to trigger incast on or off messages to the involved sources.

1. **Upon a packet arrival:** if its SYN bit is set for establishing a new TCP connection, then the current value of β for the switch port is incremented and the options information of the source VM extracted



(a) SICC framework components

(b) SICC-based data center

Fig. 2: (a) A high level view of SICC framework components' interactions which forms some form of a closed-loop control cycle. (b) A full SICC data center deployment with relations among end-hosts, switches and the controller.

Algorithm 1: SICC Application Algorithm

```
1  Function Packet_Arrival(P, src, dst)
2      if SYN_bit_set(P) then
3          β ← β + 1;
4          M[src][dst] ← P.tcpoption.mss;
5          W[src][dst] ← P.tcpoption.wndscale;
6      if FIN_bit_set(P) then
7          β ← MAX(0, β − 1);
8  Function Incast_Detection_Timeout
9      forall sw in SWITCH do
10         forall p in SWITCH_PORT do
11             Q[sw][p] ← Q[sw][p]/4 + 3×Q[sw][p]/4;
12             γ ← β[sw][p] × TCP_Icwnd + Q[sw][p];
13             if now − κ[sw][p] >= DM then
14                 if q[sw][p] ≤ (α1 × B) then
15                     forall dst in PORT_DST do
16                         forall src in DST_SRC do
17                             msg ← "INCAST OFF";
18                             send msg to src;
19             if β > 0 and γ ≥ (α2 × B) then
20                 forall dst in PORT_DST do
21                     forall src in DST_SRC do
22                         msg ← "INCAST ON";
23                         msg ← msg + W[src][dst];
24                         msg ← msg + M[src][dst];
25                         send msg to src;
26             β[sw][p] ← 0;
27     Restart Incast detection timer T;
```

from the TCP headers (i.e., the window scaling and the maximum segment size). Otherwise, if this is a packet with the FIN bit set then the current value of β for the switch port is decremented.

2. **Incast detection timer expiry:** Q_{len}^{next} indicates the minimal number of extra bytes that will be introduced into the network by the β new and existing connections. Typically each new connection starts by sending an initial congestion window-worth ($Init\_cwnd$) into the network while existing ones will maintain the same persistent (average) queue occupancy built over the course of their activity. If the buffer is expected to overflow in the next interval due to the additional traffic introduced by the new connections, then a fast proactive action must be taken to make room for the forthcoming possible incast traffic. The controller immediately sends to the hypervisor of the senders involved in the congestion situation a message to raise up their incast flag (INCAST-ON). In contrast, if the buffer occupancy is seen to drop below the incast safe threshold (i.e., 20% of the buffer size) or the time since the incast ON exceeds the expected activity time of mice flows, then we send to the involved hypervisor in the incast a message to lower down their incast flag (INCAST-OFF).

4.2 Hypervisor Window Update Algorithm

At the end end-host, the hypervisor or the vswitch are patched and modified to track for any possible incast ON/OFF messages coming from the DC controllers. The newly added function implements the receiver window rewriting to 1 MSS on the incoming ACK segments whenever incast in ON. To reach the appropriate hypervisor/vswitch, the controller uses the VMs IP address as destination, however, to prevent the hypervisor from delivering such controller messages to the VMs, the Ethernet frame is tagged with one of the unused (experimental) Ethernet types to indicate that the message carried in the frame is not a TCP/IP packet but rather an incast ON or OFF message. The hypervisor implements the following algorithm to act upon arrival of any messages from the controllers. Algorithm 2 han-

dles three type of incoming packets: incast ON, incast OFF and TCP ACK packets as follows:

Algorithm 2: SICC Hypervisor Algorithm

```
1  Function Packet_Arrival(P, src, dst)
2      if INCAST_ON_MSG(P) then
3          κ[src][dst] = True;
4          W[src][dst] = P.wndscale;
5          M[src][dst] = P.mss;
6      if INCAST_OFF_MSG(P) then
           κ[src][dst] = False ;
7      if ACK_bit_set(P) then
8          WND_Scaled = M[src][dst] >> M[src][dst];
9          if κ[src][dst] and rwnd (P) > WND_Scaled
             then
10             rwnd (P) = WND_Scaled;
11             Recalculate Internet Checksum for P;
```

1. **Incast ON:** If the received packet is identified as an "Incast ON" from the payload of the Ethernet frame. Then the hypervisor sets the incast flag to ON for this source-destination pair and extracts the attached information about the destination (i.e., the window scale shift exponent and the maximum segment size) for ACK receiver window rewriting.
2. **Incast OFF:** If the received packet is identified as an "Incast OFF", then the hypervisor resets the incast flag (to OFF) and stops ACK rewriting for this source-destination pair.
3. **TCP ACK:** If the received packet is identified as an incoming TCP ACK segment, the hypervisor checks if the incast flag for the corresponding source-destination pair is on, and starts rewriting the receiver window field to 1 MSS shifted by the window scale factor of this source-destination pair.

Setting the receive window of the ACKs to a conservative value of 1 MSS, will ensure to some extent that short query traffic (10-100KB) flows will not experience packet drops at the onset of the flow (when loss recovery via three duplicate ACK is not possible) and hence will not incur the waiting time for retransmission timeout. In addition, the incast flag is cleared as soon as the queue length drops below a predetermined threshold and/or the number of RTTs for mice to finish has expired, enabling thus elephant flows to reuse their existing congestion window values (that was simply inhibited by the receiver window rewriting) and thus restore their sending rate.

4.3 Practical Aspects of SICC Framework

SICC framework can maintain a very low in-network loss rate during incast events and enables the switch buffer to absorb sudden traffic surges while maintaining a high utilization. Therefore it can cope well with the co-existence of mice and elephants. SICC adopts a proactive recovery actions in face of the forecast incast congestion. As soon as the incoming traffic gives indication of overflowing the buffer, the receive window is shrunk to a conservative 1 MSS. Furthermore the new window is equally and temporally applied to all ongoing flows that contribute to the congestion event, meaning that all flows (mice or elephants) will receive an equal treatment during incast periods, which is the original goal of congestion control in general!.

Notice that SICC is a very simple mechanism divided among the DC controllers and end-hosts' hypervisor/vswitch with very low complexity and can be integrated easily in any network whose infrastructure is based on SDN. In addition, the window update mechanism at the hypervisor is so simple that it only requires an $O(1)$ processing per packet, as a result the additional computational overhead is insignificant for hypervisors running on DC-grade servers. SICC can also cope with Internet checksum recalculation very easily and efficiently after header modification, by applying the following straightforward one's-complement add and subtract operations on three 16-bit words: $Checksum_{new} = Checksum_{old} + rwnd_{new} - rwnd_{old}$ [19]. This also takes $O(1)$ per modified packet. In addition, since SICC is designed to deal with TCP traffic only, adding two rules to Open-Flow switches to forward a copy of SYN and FIN packets are simple operation in an SDN/Open-Flow based setup. The new rules will be a simple wildcard matching over all fields except for TCP flags which do not require per-flow information tracking at the switches, this completely conforms with the recent OpenFlow 1.5 specification [16]. Last but not least, to avoid any potential mismatch between predicted congestion in a switch buffer and actual congestion experienced in another switch buffer due to possible route changes, the forward and backward routes can be pinned down easily along the same path by the SDN controller; (notice that, unlike in wide area Internet, such route changes are very highly unlikely to happen in DCs due to path stickiness and the reliance on switches rather than routers.)

SInce SICC relies on tracking SYN packets, it may be susceptible to performance degradation when subjected to the infamous SYN flooding attacks [9]. This attack may lead the senders' window to frequently fluctuate between the current full rate and 1 MSS per RTT. This is a well-known attack that can affect the opera-

tion of any TCP flavor, including DCTCP for instance, because it disrupts TCP in general. Many proposals have suggested possible solutions to mitigate this attack [9]. SICC can leverage FloodGuard [22] which implements an efficient, lightweight and protocol-independent defense framework for SDN networks. It was shown that it is effective in mitigating flooding attacks while adding only negligible overhead into the SDN framework.

5 Simulation and Performance Analysis

In this section, we compare the performance of SICC framework to TCP-DropTail, TCP-RED, in to two alternative mechanisms that rely on modifying the switch, in RWNDQ [2] or modifying the TCP protocol in the VM in DCTCP [6]. For SICC, the values of α are chosen based only on the level of queue occupancy that signals end of incast, T_i is set to a value larger than the end-to-end average RTT. In the simulations, we set α to 20% of the buffer size, T_i to 5 ms (i.e., 10 times the RTT). DCTCP parameters are set according to the recommended settings in [6] with $K = 0.17$. We use the network simulator ns2 version 2.35 [14], which we have extended to include SICC framework as well as TCP flow control. For DCTCP, we use a patch for ns2.35 available from the authors [5] and for proper operation, ECN-bit capability is enabled in the switch and TCP sender/receiver. We use in our simulation experiments high speed links of 1 Gb/s for sending stations and the bottleneck links, average RTT of 500μs and the MinRTO of 200ms which is the default in Linux TCP implementations. The buffer size is set to 83 packets (i.e., 125 KBytes) while the IP data packet size is 1500 bytes.

5.1 Single-rooted Topology Simulation

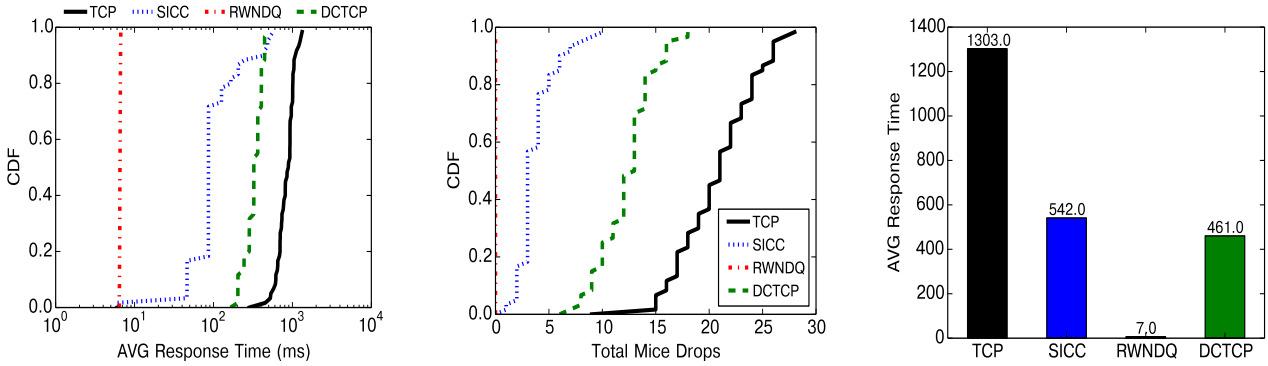
First, we use a single-rooted (dumbbell) topology and run the experiments for a period of 5 sec. Using 80 FTP flows, we simulate two scenarios to mimic synchronized incast traffic competing with long-lived flows at the same time. In the first scenario, we simulate an elephant-to-mice ratio of 1:3 which is close to the reported ratio of elephants to mice in data centers [6, 8]. We rerun the simulation but increase the share for elephants to the ratio of 3:1 to examine how SICC would respond in situations where the network is highly loaded with long-lived (background) traffic. While elephants keep sending at full possible speed, during the whole simulation period, mice finish very quickly then close the connection and reopen a new one at the beginning of each second (i.e., 5 incast epochs). To ensure a relatively tight synchronization between mice flows, mice flows start in a random order with average inter-start time of one packet transmission time. Each mouse flow sends 10 KBytes of data then halts until the start of the next epoch.

Fig. 3 shows the CDF distributions of the mean and average 99^{th}-percentile of the flow completion time (FCT) for mice, the packet drops experienced by mice and the average goodput obtained by elephant flows. Fig. 3a show that SICC can improve mice flow completion time on average with lower variation in the response times, achieving a performance close to DCTCP. RWNDQ which is switch based improves even further due to its agility in setting the fair-share of the flows. SICC can improve TCP's performance yet requires no modification to the communication end-points nor the switches. Fig. 3b shows the total cumulative mice packets drops during the 5 epochs at the bottleneck link. This gives an insight on how SICC is helping mice to achieve faster FCT by reducing packet drops, thus allowing TCP to avoid the huge penalty of waiting for timeout. Finally, Fig. 3c shows the average (99th-percentile) completion time of the 5 epochs where clearly SICC is helping TCP to achieve faster FCT even on the tail giving an advantage for co-flows type of applications without the need for fully fledged scheduling.

In the 1:1 elephants-to-mice ratio case, Fig. 4a and 4b show that SICC can still improve the mice average FCT with low standard deviation. The FCT in this case is improved for the majority of mice flows compared to DCTCP due SICC's ability to quench elephants during incast. The tail FCT of mice is also improved under SICC as suggested by Fig. 4c due to the reduced mice packet drops at the bottleneck by throttling the elephants to make room for mice.

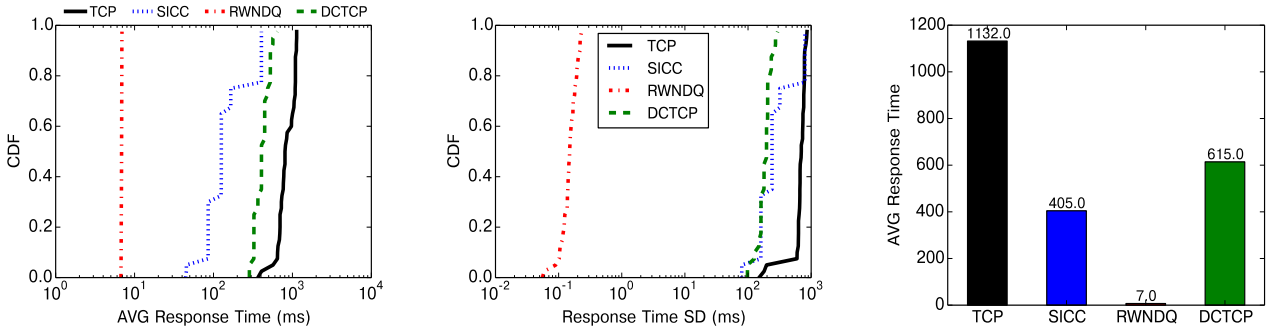
In the 3:1 elephants-to-mice ratio case, Fig. 5a and 5b show that SICC can still improve the mice average FCT with low variation with significant improvement over DCTCP. In this highly loaded scenario, DCTCP performs the worst close to (or even worse than) TCP while RWNDQ being switch-based, still gives the best performance. Finally, Fig. 5c shows that SICC is able to reduce TCP's drop probability at the bottleneck and hence improve the tail FCT of mice flows.

Elephant Flows Performance: Fig. 6 clearly shows that in all these scenarios SICC does not degrade the performance of elephant flows and it has nearly no impact on the achieved goodput compared to TCP. This outcome is attributed to the temporary involvement of SICC during incast activity and its ability to restore elephant flows' original sending rate quickly.



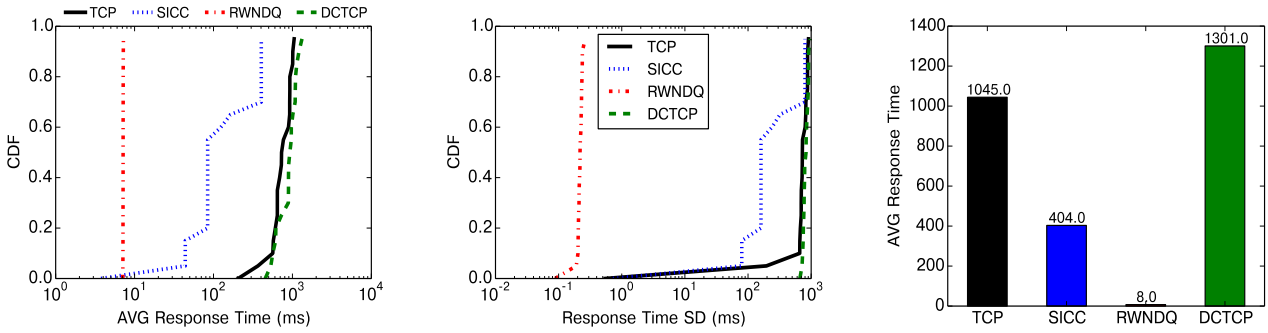
(a) Average FCT for mice (b) Mice packet drops (c) Average 99^{th}% FCT

Fig. 3: Performance metrics of mice flows for TCP, SICC, RWNDQ and DCTCP in 1:3 ratio scenario.



(a) Average FCT for mice (b) AFCT SD for mice (c) Average 99^{th}% FCT

Fig. 4: Performance metrics of mice flows for TCP, SICC, RWNDQ and DCTCP in 1:1 ratio scenario.

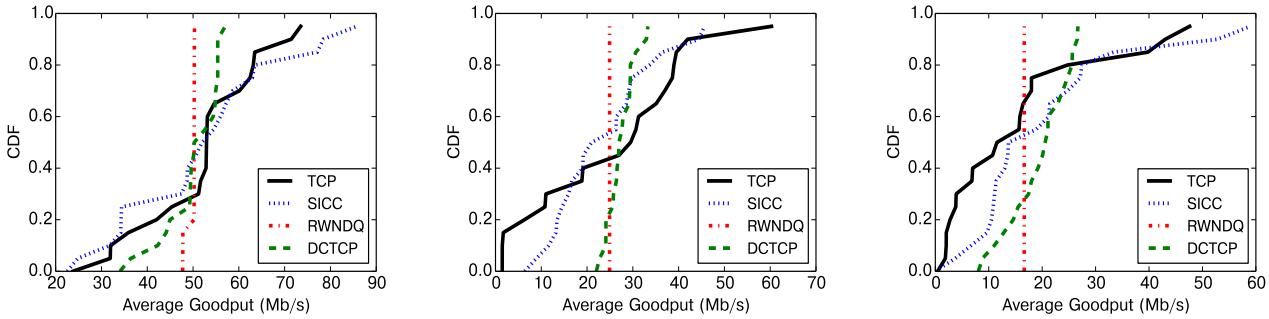


(a) Average FCT for mice (b) AFCT SD for mice (c) Average 99^{th}% FCT

Fig. 5: Performance metrics of mice flows for TCP, SICC, RWNDQ and DCTCP in 3:1 ratio scenario.

5.2 Fat-tree Datacenter Topology Simulation

We have also created a fat-tree like topology as show in Fig. 7 with 1 core, 2 aggregation, and 3 ToR switches each connecting 48 servers to mimic the topologies used in real data center environment. We positioned in the third rack an aggregation server which receives data from all 144 server in this network. We use in our simulation experiments links of 10Gb/s in from core switch to aggregation switches, 5Gb/s from aggregation switches to ToR switches and 1Gb/s from ToR switches to Servers. This setup creates an over-subscription ratio of 1:24 at the ToR level (a moderate value to what has been reported in todays DCs with values up to 1:80 of over-subscription). We use a propagation delays of 25μs per link and a MinRTO in the VM of 200ms. In this scenario, the elephants communicates as follows: Rack1→Rack3, Rack2→ Rack3 and Rack3→Rack1. Mice communication pattern defines Racks 1, 2 and 3 to be the workers



(a) 1:3 elephant-to-mice ratio (b) 1:1 elephant-to-mice ratio (c) 3:1 elephant-to-mice ratio

Fig. 6: Performance metrics of elaphant flows for TCP, SICC, RWNDQ and DCTCP.

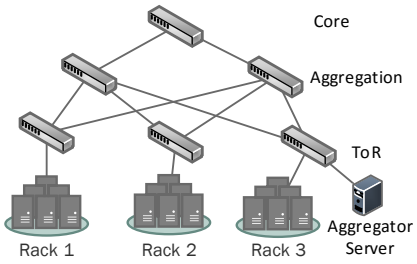


Fig. 7: A fat tree topology connecting 145 servers.

who are sending results back to the aggregation server in rack 3 over 5 epochs during the simulation.

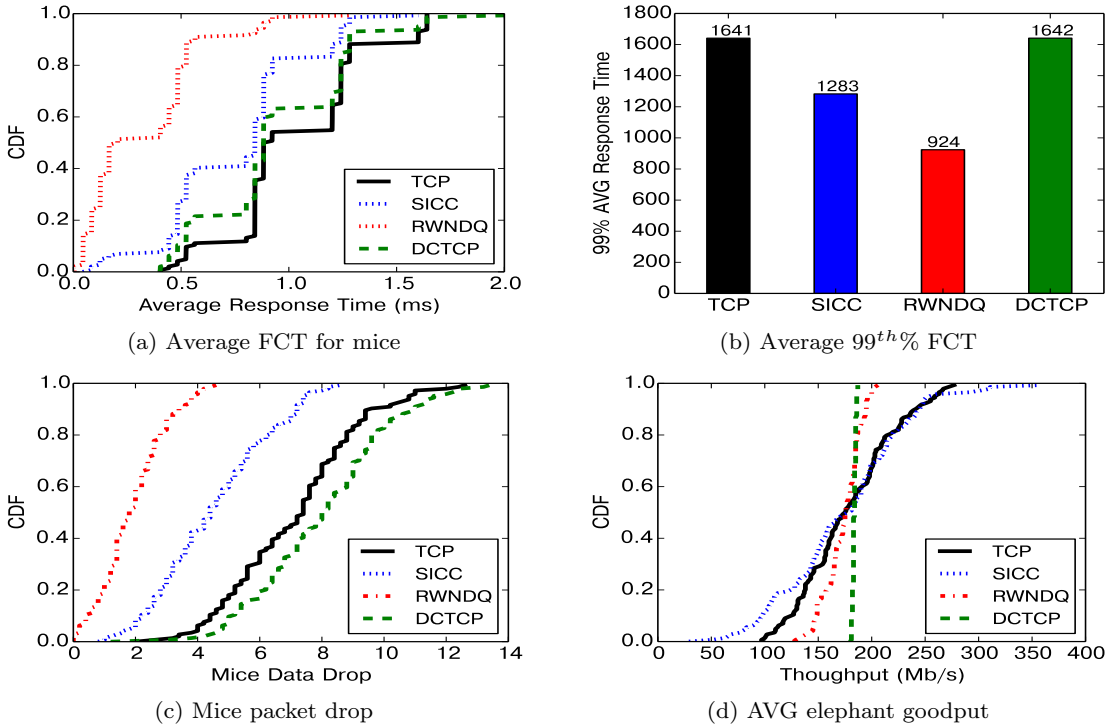
Fig. 8 shows the results for this scenario. SICC is able to improve incast flows FCT compared to TCP and achieves comparable performance as DCTCP with nearly no impact on elephants throughput. As expected RWNDQ outperforms all schemes due to its accurate estimation of the fair-share at the switch. The reduced FCT is mainly attributed to the reduced packet drops of short-lived mice flows.

We rerun the simulation but this time in a larger data center setup with 3 aggregation and 6 ToR switches (i.e., 6 Racks) leading to a network of (6×28) 288 servers. Elephant flows are defined as Rack(1,2)→Rack(3,4), Rack(3,4)→Rack(5,6) and Rack(5,6)→Rack(1,2). Fig. 9 shows the results. SICC and RWNDQ can improve TCP's performance and both achieve better performance than DCTCP in a larger over-subscribed data center. The improvement is mainly attributed to the reduced mice packet average drop rate and hence average number of failed flows for SICC are reduced as shown in Fig 9a's legend.

5.3 Sensitivity of SICC to the monitoring interval

We repeat the single-rooted experiment with 1:3 elephant-to-mice ratio for SICC while varying the monitoring interval over which we read the queue occupancy and based on which we detect incast events. In this simulation, we ran the simulation for values of T_i as described in SICC Algorithm 1 normalized to the RTT value (i.e., 100 μs). To cover a wide range of values we simulated T_i to be (1, 2, 10, 20, 25, 30, 50, 100) times the RTT. Fig. 10 shows the distributions of the mean and variance of FCT for mice, average (99th-percentile) completion time of mice and the average goodput for elephant flows. Fig. 10d implies that SICC's monitoring interval does not affect the achieved goodput of TCP but it would affect the efficiency of SICC's incast detection ability. Fig. 10a, 10b and 10c show that SICC can still achieve a good performance, even with a monitoring interval 25 times longer than the RTT in the network. This analysis suggests that a value of $\approx$ 1-25 RTT in the network would be sufficient. In typical datacenters, with a minimum RTT of 200-250μs, this translates to reading the queue occupancy once every 4-7ms which seems an acceptable probing interval for SDN controllers. This justifies the choice of a monitoring interval of 10 times the RTT in the previous simulations.

We also did a sensitivity analysis through multiple simulations (not shown here) on the values of α and $safe\_thr$ parameter, we found that SICC is not sensitive to these values. In terms of bandwidth overhead, we can quantify the amount of bytes for communicating the queue size information from the SDN switches to the controller(s). Assume we have a network consisting of 1000 switches (48 ports per switch) and 1 controller and assuming a probing interval of 5 ms then a TCP message of size 48-port*2 bytes/queuesize in the payload + 54 bytes for TCP, IP and ETH headers yields a 150 bytes message per switch. In total, for the 1000 switches the controller would receive 150Kbytes every 5ms, which translates a bandwidth 240 Mbit/s. We believe this is reasonable bandwidth utilization for communication overhead between the switches and the controller with respect to the performance gain for the



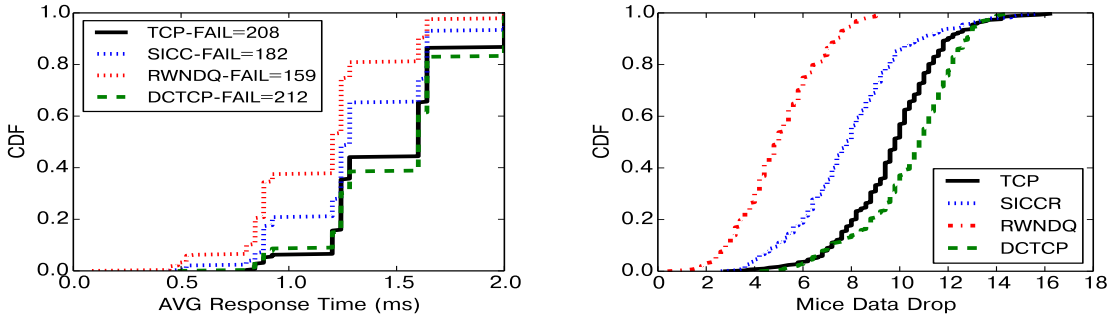
(a) Average FCT for mice

(b) Average 99^{th}% FCT

(c) Mice packet drop

(d) AVG elephant goodput

Fig. 8: Performance metrics of TCP, SICC, RWNDQ and DCTCP in small fat-tree topology of 144 servers.



(a) Average FCT for mice

(b) AVG mice data drop

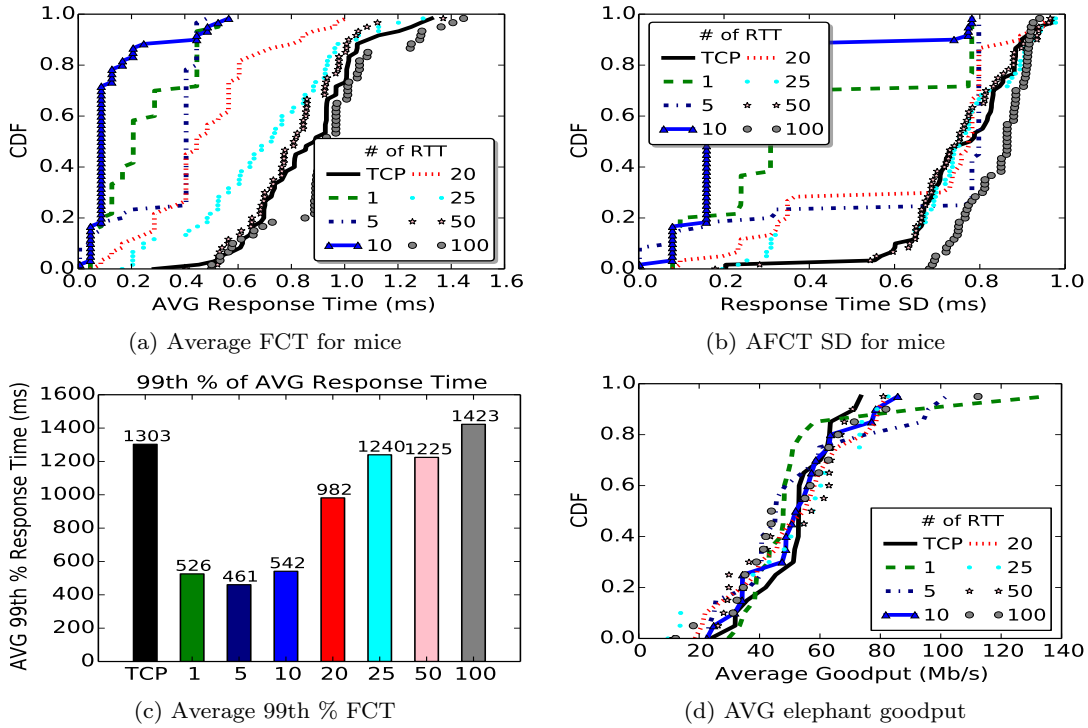
Fig. 9: Performance metrics of TCP, SICC, RWNDQ and DCTCP in larger fat-tree topology of 288 servers.

majority of incast flows in datacenters. In addition, in most of current SDN setups, control plane signaling is out-of-band [18].

6 Testbed implementation of SICC framework

We further investigate the implementation of SICC as an application program integrated with the Ryu controller [20] for experimentation in a real-testbed. SICC was implemented in python programming language as a separate applications to run along with any python-based SDN controller. We also patched the Kernel datapath modules of Openvswitch (OvS) [17] with the window update functions described in subsection 4.2. We added the update function in the processing pipeline of the packets that pass through the datapath of OvS. In a virtualized environment, OvS can process the traffic for inter-VM, Intra-Host and Inter-Host communications. This is an efficient way of deploying the window update function on the host at the hypervisor/vswitch level by only applying a patch and recompiling the running kernel module, making it easily deployable in today's production DCs with minimal impact on the traffic and without any need for a complete shutdown[2].

[2] Typical throughput of internal networking stack is 50-100 Gb/s, which is fast enough to handle 10's of concurrent



(a) Average FCT for mice

(b) AFCT SD for mice

(c) Average 99th % FCT

(d) AVG elephant goodput

Fig. 10: SICC with variable queue monitoring interval.

6.1 Testbed Setup

For experimenting with our SICC framework, we set up a testbed as shown in Fig. 11. All machines' internal and the outgoing physical ports are connected to the patched OvS on the end-host. We have 4 racks: rack 1, 2 and 3 are senders and rack 4 is receiver. each rack has 14 servers (however we run experiments on 7 of them) all installed with Ubuntu Server 14.04 LTS running kernel version (3.16) and are connected to the ToR switch through 1 Gb/s links. The core switch in the testbed is OvS switch running on another server (i.e., the 8^{th} server in one of the racks) which are employed with 4-ports server NIC. The OvS are setup to enable matching on the TCP flags [16][3]. Similarly, the VMs are installed with the iperf program [10] for creating elephant flows and the Apache web server hosting a single webpage **"index.html"** of size 11.5KB for creating mice flows. We setup different scenarios to reproduce both incast and buffer-bloating situations with bottleneck link in the network as shown in Fig. 11. The senders are created by creating multiple virtual ports on the OvS at the end-hosts and binding an iperf or an Apache client/server process to each vport which allow us to create scenarios with large number of flows in the network. In the experiments we have set the monitoring interval (i.e., T_i) to a conservative value of 50 ms whereas the network RTT ranges from 300μs without queuing and up to 1-2 ms with excessive queuing.

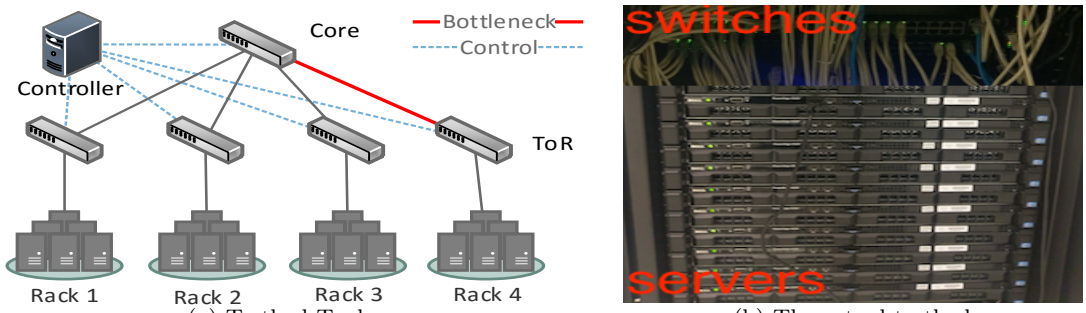
6.2 Experimental Results

The goals of the experiments are to: *i)* Show that TCP can support many more connections and maintain high link utilization with the introduction of SICC framework; *ii)* Verify whether SICC can help TCP overcome incast congestion situations in the network by improving mice completion time; *iii)* Study SICC's impact on the achieved throughput of elephants.

We run an incast with buffer-bloating scenario, in which mice traffic compete with elephant flows, to see if SICC can help mice flow's average FCT during incast period. We first generate 7 synchronized iperf elephant connections from sender racks continuously sending for 30 secs resulting in 21 ($7 \times 3 = 21$) elephants at the bottleneck. We use Apache benchmark [7] to request **"index.html"** webpage from each of the 7 web servers at each of the sending racks ($7 \times 6 \times 3 = 126$ in total) running on the same machines where iperf are sending. Note that, we run Apache benchmark, at the 15^{th}sec,

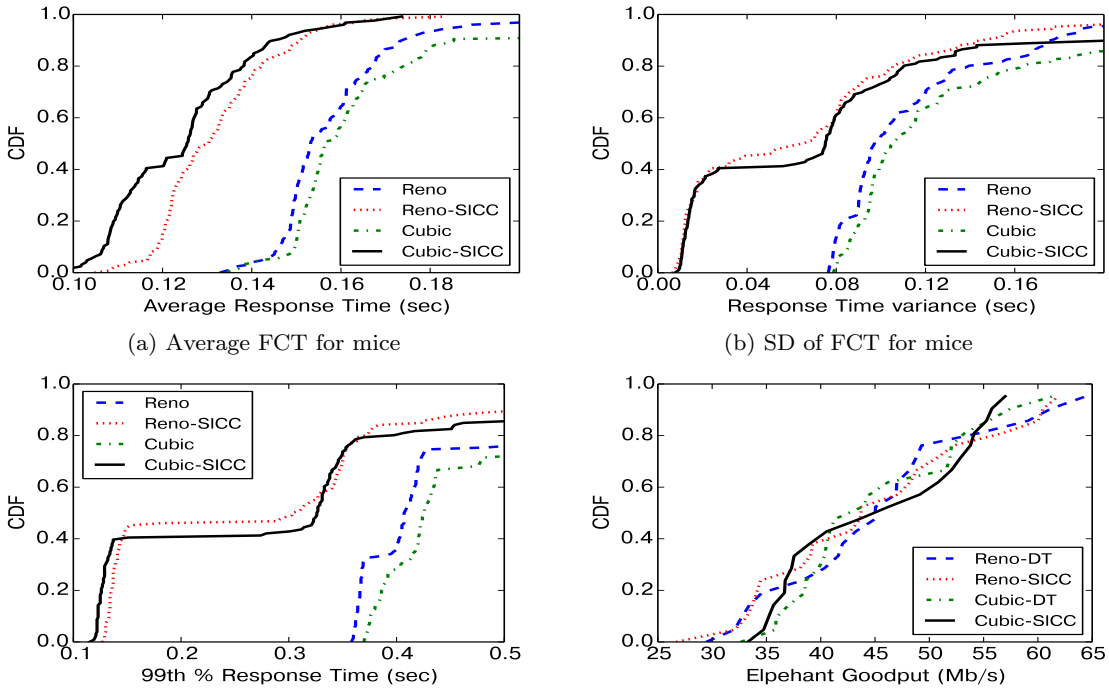
VMs sharing a single or few physical links. Hence, the window update function added to the vswitch would not hog the CPU and hence the achievable throughput.

[3] The hardware switch running OF-DPA does not follow OF1.5 specifications



(a) Testbed Toplogy (b) The actual testbed

Fig. 11: A real SDN testbed for experimenting with SICC framework using Ryu-Controller, OpenFlow switches and OpenvSwitch



(a) Average FCT for mice (b) SD of FCT for mice

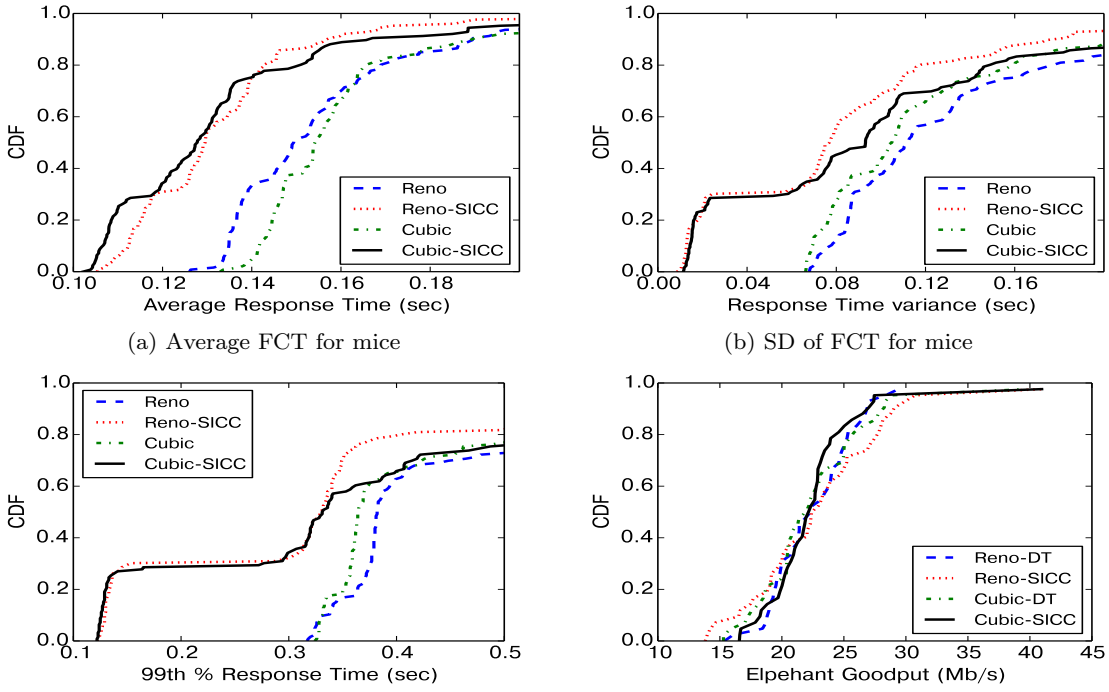
(c) 99th % of FCT for mice (d) AVG elephant goodput

Fig. 12: TCP-SICC vs. TCP-Droptail: 126 mice incast-like flows competing with 21 long-lived elephants

requesting the webpage 10 times then it reports different statistics over the 10 requests. We repeated the previous experiment but in this case using TCP cubic as the congestion control. Fig. 12 shows that, in both cases, SICC achieves a good balance in meeting the conflicting requirements of elephants and mice. Specifically, Fig. 12d shows that the long-lived elephants are not affected by SICC's inhibition of their sending rate for a very short period of time after which they restore their previous rates. However, the competing mice flows benefit greatly under SICC by achieving a smaller FCT on average with a smaller standard deviation compared to TCP as shown in Fig. 12d and 12b. In addition, as SICC efficiently detects the incast and proactively throttles the elephants, it can decrease the flow completion time even on the tail (i.e., 99th percentile) as shown in Fig. 12c.

We repeated the experiment but with 2 iperf flows per sender leading to 42 elephant flows ($7 \times 3 \times 2 = 42$). Fig. 13 shows that SICC still achieves a reasonable performance improvement for both TCP NewReno and TCP Cubic. Fig. 13d shows that long-lived elephant flows are not affected by SICC. Fig. 13a shows that mice flows still benefit under SICC even in a situation where buffers are pressured by the large number of elephants.

In summary the experimental results reinforce the results obtained in the simulation. In particular, they show that:



(a) Average FCT for mice

(b) SD of FCT for mice

(c) 99th % of FCT for mice

(d) AVG elephant goodput

Fig. 13: TCP-SICC vs. TCP-DropTail: 126 mice incast-like flows competing with 42 long-lived elephants

1. SICC helps in reducing mice traffic latency and maintains a high throughput for elephants.
2. SICC handles incast events in low and high load scenarios while nearly fully utilizing the communication links.
3. SICC achieves all this without the need for any TCP stack alternation and/or new switch-based scheme.

7 Conclusion and future work

In this paper, we proposed a SDN-based congestion control framework to support and help reduce the completion time of short-lived incast flows, that are known to constitute the majority of flows in data centers. Our framework SICC mainly relies on the SDN controller to monitor the SYN/FIN packets arrivals along with reading over regular intervals the OpenFlow switch queue occupancy to infer the start of incast-traffic epochs before they start sending data into the network. SICC was shown via ns2 simulations and testbed experiments to improve the flow completion times for incast traffic without impairing the throughput of elephant flows. Our SICC framework is also shown to be simple, practical, easily deployable and also it meets all its design requirements. A number of detailed simulations showed that SICC can achieve its goals efficiently while outperforming the most prominent alternative approaches.

Last but not least, knowing that in most public data centers, it is beneficial to both the operator and tenants if the congestion control framework is deployable without making any changes to the TCP sender and/or receiver nor replacement of the in-place commodity hardware switching. SICC's main contribution is to adhere to such principle and achieve its performance improvement without modifying the TCP algorithms nor the networking hardware, enabling thus a quick and true deployment potential in real operation-critical data center networks. Further testing of SICC in an operational environment with realistic workloads and scale is necessary.

Acknowledgements This work is supported in part under Grants: HKPFS PF12-16707, REC14EG03 and FSGRF13EG14.

References

1. Abdelmoniem, A.M., Bensaou, B.: Efficient switch-assisted congestion control for data centers: an implementation and evaluation. In: 34th IEEE International Performance Computing and Communications Conference (IPCCC15) (2015)
2. Abdelmoniem, A.M., Bensaou, B.: Incast-Aware Switch-Assisted TCP congestion control for data

centers. In: IEEE Global Communications Conference (GlobeCom15) (2015)
3. Abdelmoniem, A.M., Bensaou, B.: Reconciling mice and elephants in data center networks. In: Proceedings of IEEE CloudNet (2015)
4. Abdelmoniem, A.M., Bensaou, B., Abu, A.J.: SICC: SDN-based Incast Congestion Control for Data Centers. In: IEEE International Conference on Communications (ICC17) (2017)
5. Alizadeh, M.: Data Center TCP (DCTCP). http://simula.stanford.edu/ alizade/Site/DCTCP.html
6. Alizadeh, M., Greenberg, A., Maltz, D.A., Padhye, J., Patel, P., Prabhakar, B., Sengupta, S., Sridharan, M.: Data center TCP (DCTCP). ACM SIGCOMM CCR **40**, 63–74 (2010). DOI 10.1145/1851275.1851192
7. Apache.org: Apache HTTP server benchmarking tool. Http://httpd.apache.org/docs/2.2/programs/ab.html
8. Benson, T., Akella, A., Maltz, D.a.: Network traffic characteristics of data centers in the wild. In: Proceedings of ACM SIGCOMM (2010). DOI 10.1145/1879141.1879175
9. Eddy, W.: RFC 4987 - TCP SYN Flooding Attacks and Common Mitigations (2007). Https://tools.ietf.org/html/rfc4987
10. iperf: The TCP/UDP Bandwidth Measurement Tool. Https://iperf.fr/
11. Kandula, S., Sengupta, S., Greenberg, A., Patel, P., Chaiken, R.: The nature of data center traffic. In: Proceedings of IMC (2009). DOI 10.1145/1644893.1644918
12. Lu, Y., Zhu, S.: SDN-based TCP Congestion Control in Data Center Networks. In: Proceedings of IEEE IPCCC (2015)
13. Mckeown, N., Anderson, T., Peterson, L., Rexford, J., Shenker, S., Louis, S.: OpenFlow : Enabling Innovation in Campus Networks. ACM SIGCOMM CCR **38**, 69–74 (2008). DOI 10.1145/1355734.1355746
14. NS2: The network simulator ns-2 project. Http://www.isi.edu/nsnam/ns
15. Open Networking Foundation: SDN Architecture Overview. Tech. rep., Open Networking Foundation (2013)
16. opennetworking.org: OpenFlow v1.5 Specification. Https://www.opennetworking.org/sdn-resources/openflow
17. OpenvSwitch.org: Open Virtual Switch project. Http://openvswitch.org/
18. Panda, A., Scott, C., Ghodsi, A., Koponen, T., Shenker, S.: CAP for networks. In: Proceedings of HotSDN workshop (2013)
19. Rijsinghani, A.: RFC 1624 - Computation of the Internet Checksum via Incremental Update (1994). Https://tools.ietf.org/html/rfc1624
20. Ryu Framework Community: Ryu: a component-based software defined networking controller. Http://osrg.github.io/ryu/
21. Vasudevan, V., Phanishayee, A., Shah, H., Krevat, E., Andersen, D.G., Ganger, G.R., Gibson, G.A., Mueller, B.: Safe and effective fine-grained TCP retransmissions for datacenter communication. ACM SIGCOMM CCR **39** (2009). DOI 10.1145/1594977.1592604
22. Wang, H., Xu, L., , Gu, G.: Floodguard: A dos attack prevention extension in software-defined networks. In: IEEE/IFIP Conference on Dependable Systems and Networks (2015)
23. Wu, H., Feng, Z., Guo, C., Zhang, Y.: ICTCP: Incast congestion control for TCP in data-center networks. IEEE/ACM Transactions on Networking **21**, 345–358 (2013). DOI 10.1109/TNET.2012.2197411