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), the presence of long lived TCP flows tends to bloat the switch buffers. As a consequence, short-lived TCP-incast traffic suffers repeated losses that often lead to loss recovery via timeout. Because the minimum retransmission timeout (minRTO) in most TCP implementations is fixed to around 200ms, interactive applications that often generate short-lived incast traffic tend to suffer unnecessarily long delays waiting for the timeout to elapse. 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 to achieve the same result 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 or TCP protocol. In this approach, we rely on the programmable nature of SDN switches and design an SDN-based Incast Congestion Control (SICC) framework that uses an SDN network application in the controller and a shim-layer in the host hypervisor to mitigate incast congestion. We demonstrate the performance gains of the proposed scheme via real deployment in a small-scale testbed as well as ns2 simulation experiments in networks of various sizes and settings.

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
*Corresponding Author. The source code, simulation and experiments is 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 in applications that generate a large number of traffic flows, with varying characteristics and requirements, that range from large groups of barrier-synchronized, short-lived, time-sensitive flows, to long-lived, time-insensitive, throughput inclined flows. For example, often traffic flows that originate from partition-aggregate applications (e.g., web query, map-reduce and so on) fall into the first category, due to time constraints imposed on them by the need for interactivity. In contrast, traffic flows that originate from backups and virtual machine migration, often fall into the second category. In the sequel we refer to the former as mice flows and to the latter as elephants. Recent studies [10, 16, 28] show that in practice elephant flows take the lion's share of the traffic volume in DCNs, nevertheless, such networks teem with many more mice flows.

DCNs are structured to provide a high bandwidth with low latency, as such, unlike in the Internet, the traffic "inertia"[1] in such networks is small, thus, the amount of buffer space required to absorb bursts of traffic is also small. As a consequence, DCNs mostly use Ethernet switches with small buffers (instead of routers) to interconnect their servers. However, in the

[1] We define the traffic inertia of a flow as the worst case amount of traffic that the flow can inject in the network before congestion can be detected. This is typically the maximum amount of traffic that could be lost before the loss is detected and recovery is invoked.

presence of such small buffers, the sudden surge of synchronized incast-TCP flows still results in severe congestion events. In particular, in the presence of elephants the shared buffer is bloated, leaving little room for the incoming synchronized small flows. TCP was not designed to deal with such complex congestion events, as it is agnostic to the latency requirements of mice flows, as well as to the composite nature of the application data (i.e., the existence of many synchronize flows that contribute to the same application). Therefore it does not handle them effectively as shown in many recent works [7, 8, 11, 28, 29].

Recent works in [1, 3, 7, 18, 27, 28] represent the typical set of approaches used to addressed this issue. Many such works adopt an end-to-end TCP-AQM as a means of maintaining a small queue in the switches, ensuring thus a high throughput for elephants while maintaining enough empty buffer space to absorb mice burst arrivals. For instance, DCTCP [7] modifies TCP reaction to ECN marks to cut the congestion window (*cwnd*) in proportion to the number of marks received per RTT, and 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 Center (DC), in our earlier work we proposed RWNDQ [3], 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*) upon congestion build up, to impose this fair share on all the sources regardless of their nature (mice or elephants). Both approaches involve modification of either the TCP stack in the guest VMs for the former or the switch software in the latter. This makes them less appealing for immediate deployment in large DCNs due to additional restrictions and cost they impose.

Software Defined Networking (SDN) [5, 13, 21] was recently adopted as a router and switch design approach that separates the control functions from the data-path, outsourcing the former to a dedicated central controller(s) with a global-view of the network state. OpenFlow [19] is currently the dominant standard interface between the controller and the data-path. This approach 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 a simple application software. To avoid modifying the TCP stack or the switch, in this paper, we invoke the programming ability of SDN switches to design an SDN-based incast-TCP congestion controller (SICC) that consists of an SDN network application in the SDN controller and a shim-layer in the host server hypervisor.

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 be applied to small scale private data centers. In particular, in most public cloud services, tenants rent virtual machine (VM) instances and can upload their own operating system images to their VMs. In addition, tenant can modify/fine-tune their protocol stack as needed to achieve better performance. In contrast, modifying the switch hardware is feasible from the cloud provider perspective, however, it remains unappealing as it is a costly alternative. Therefore, in our design approach, our target is to emerge with a solution that encompasses the following principles: (R1) Effectiveness (i.e., the solution must effectively handle the problem of incast congestion by significantly improving the flow completion time (FCT) of mice flows, without degrading dramatically the throughput of elephant flows); (R2) Scalability (i.e., the solution must avoid modifying the TCP sender/receiver protocol, nor alter the hardware switch); and finally, (R3) Simplicity and Practicality (i.e., the solution must be immediately and easily deployable via simple software patches in existing DCNs that support SDN).

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 such congestion. This can be done via programming in SDN, by implementing an SDN control application (to estimate congestion and notify the hypervisors) and a shim layer in the hypervisor modules to actually apply traffic control underneath the VMs.

1.2 Summary of Contributions

1. We explore the prospect of using SDN paradigm to design congestion control schemes 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 hypervisors.
3. We implement the proposed framework and evaluate its performance via simulation and in 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 [27], 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 with microsecond timer granularity. The so-called DCTCP [7] proposed to modify 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 [28] 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 have 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 quench elephants and as a result make 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 modification.
4. **SDN-based**: SDTCP [18] 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 by 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. In another work, OTCP [15] addresses the problem of incast congestion by means of computing and adjusting certain environment-specific congestion control parameters based on centrally available network properties.

3 The Proposed Methodology

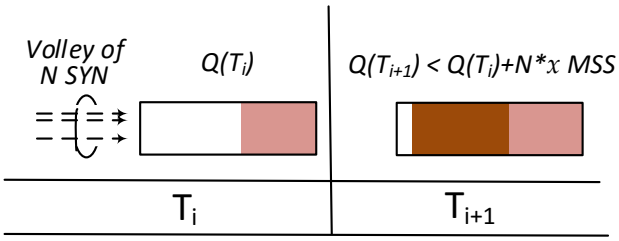


Fig. 1: SICC Idea Rationale

The basic idea behind the proposed SICC framework is illustrated in Fig.1. Considering the buffer occupancy at an outgoing port of a switch, assuming that mice flows are short-lived, they will mainly contribute to the queue variation in the switch port buffer, in contrast, the persistent queue length is mostly due to the contribution of elephant flows. 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 (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, where x is the initial window size of TCP. To control congestion, one may simply inhibit the congestion window and rely on flow control whenever $Q(T_{i+1})$ is expected to exceed the buffer size. For instance, during such events each flow might be allowed to send only 1 MSS per RTT. To ensure a high link utilization, flow control is turned off whenever the queue drains enough, reaching a given low threshold, upon which the sources recover the use of their already acquired congestion window sizes. The underlying principle with this approach is to achieve fairness in the short terms among ongoing flows (mice and

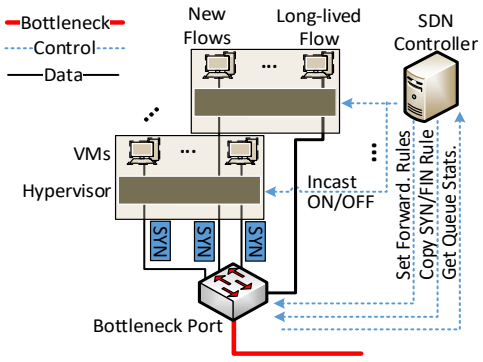


Fig. 2: A high level view of SICC framework components' interactions which forms some form of a closed-loop control cycle.

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 variant, the TCP source sending rate is determined by the sender window $swnd = \min(rwnd, cwnd)$, where $rwnd$ and $cwnd$, are 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 happen in the hypervisor below the VMs 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 the incast mode, 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 SDN controller uses rough estimates by simply counting the accumulated number N of TCP segments with a SYN-ACK bit, deducting the number of TCP segments with the FIN bit set; in the worst case, this estimate results in a conservative estimate of the predicted queue length. Without loss of generality, in the sequel we will consider the value of the initial TCP congestion window (x) to be 1 MSS.

Fig. 2 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) The hypervisor or virtual switch (vswitch) SICC monitoring module intercepts such messages; upon receipt of incast ON message destined to a certain VM, all incoming ACK packets for that VM are intercepted and their $rwnd$ values are rewritten, until an incast OFF message is received later or the average duration of typical mice (short-lived) 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.

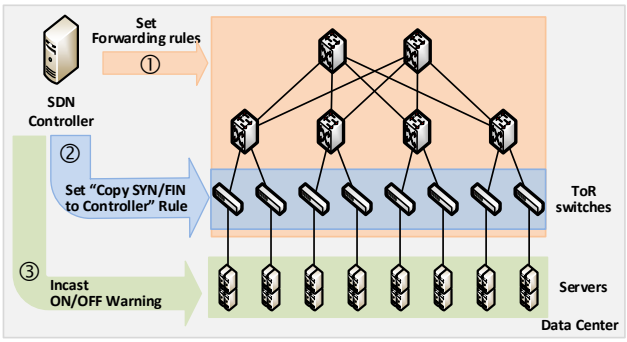


Fig. 3: A full SICC-based data center deployment with relations among end-hosts, switches and the controller.

Fig. 3 illustrates a possible deployment scenario of our proposed SICC framework in a SDN based data centers. All the switches in the DC are SDN-enabled.

The controller controls all the switches in the data center and is by default responsible for setting the forwarding rules in the switches of the data center. The controller sets the necessary rules in all the Top of Rack (ToR) switches to intercept any TCP SYN segments. As a result, the controller is able to track TCP connections and pin-down their paths by setting new forwarding rules (or merging them into existing aggregates) in the DCN switches. By also intercepting the FIN segments in the ToR switches, the controller is also able to withdraw routing rules from the switches as necessary. In addition the end-hosts' hypervisors/vswitches are patched to receive and process the incast warning messages originating from the central controller.

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	The controller monitoring interval
α_1	Queue threshold to turn OFF Incast
α_2	Queue threshold to turn ON Incast
DM	Average duration for mice flows to finish
List Objects	Description
SW	List of the controlled SDN switches
SW_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
WS	Window scale of source-destination pair
M	Maximum segment size of source-destination pair
γ	The new predicted traffic after incast arrival
β	Coarsely estimated differential of new connections
κ	Boolean true if incast is ON
IW	The dominant initial congestion (e.g., 10 MSS)

4.1 SDN Network Application for Incast Detection

The network application communicates with the central controller via the north-bound API in order to set OpenFlow rules at OpenFlow-enabled 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 (e.g., maximum segment size and receiver window scaling value). Such information is stored in source-destination-based hash tables to be used 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:

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_Timeout_Handler
9      forall sw in SW do
10         forall p in SW_PORT do
11             Q[sw][p] ← Q[sw][p]/4 + 3×Q[sw][p]/4;
12             γ ← β[sw][p] × IW + 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;
```

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 the 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 are extracted 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 timeout handler:** γ 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 (i.e., IW) worth of packets 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(s) 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 the controller sends to the involved hypervisor(s) a message to lower down their incast flag (INCAST-OFF).

4.2 Hypervisor Window Update Algorithm

At the end-host, the hypervisors or the vswitches are patched and modified to process any incoming incast ON/OFF raw messages coming from the SDN application. 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 arriving messages from the controllers. Algorithm 2 handles three type of incoming packets: incast ON, incast OFF and TCP ACK packets as follows:

1. **Incast ON:** If the received packet is identified as an "Incast ON" from the payload of the Ethernet frame. Then the hypervisor sets in the VM-to-VM table the incast flag field (i.e., κ) to ON for this source and destination pair. Then, the hypervisor extracts and updates the relevant information in the flow table about the destination (i.e., the window scale shift exponent and the maximum segment size) to be able to update the receiver window field upon ACKs arrival.
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 and 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.

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          WS[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] >>
             WS[src][dst];
9          if κ[src][dst] and rwnd(P) > WND_Scaled
             then
10             rwnd(P) ← WND_Scaled;
11             Recalculate Internet Checksum for P;
```

Setting the receive window of the ACKs to a conservative value of 1 MSS, will ensure to some extent that short query traffic flows (i.e., those of size 10-100KB) will not experience packet drops at the onset of the transfer (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 re-use 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, especially with the introduction of scalable control approaches in SDN platforms [17]. 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 one of the original goals of congestion control in general.

Notice that SICC is a very simple mechanism divided among the DC controllers and the hypervisor or 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. 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: $Checksum_{new} = Checksum_{old} + rwnd_{new} - rwnd_{old}$ [25]. This also takes $O(1)$ per modified packet. In addition, since SICC is designed to deal with TCP traffic only, adding two rules to OpenFlow switches to forward a copy of SYN and FIN packets are simple operation in an SDN/OpenFlow based setup. The new rules will be a simple wildcard filter 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 [22]. 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.)

5 Simulation and Performance Analysis

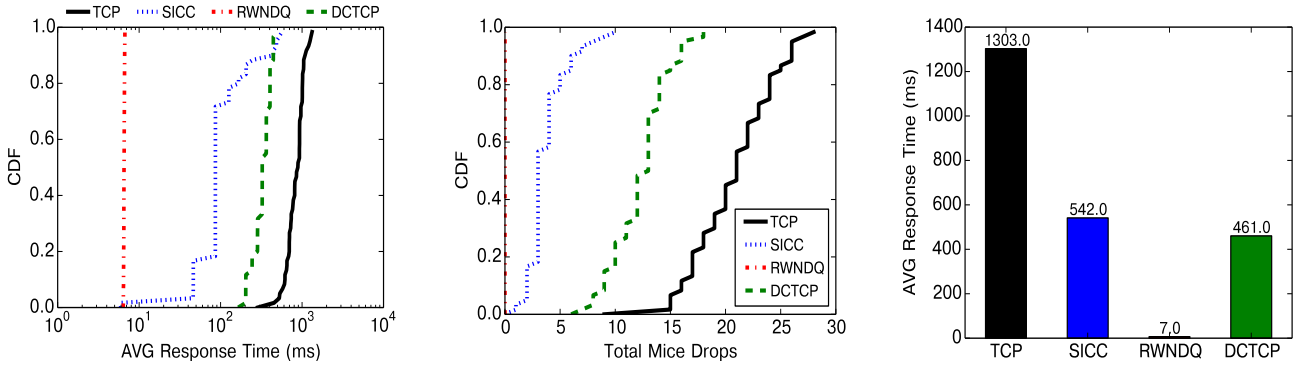
We have tested the performance of SICC via simulation in various network topologies of different scales, then to further test the practical feasibility of SICC we have implemented it in a small testbed. In this section, we report the results from our simulation study, and delay the implementation results to the next section.

We carried out several simulations in which we compared the performance of TCP/SICC to that of *i)* TCP with RED AQM, with ECN marking enabled; *ii)* the RWNDQ algorithm [2], that modifies the switch algorithm to achieve fairness; and *iii)* the DCTCP algorithm [7], that modifies/replaces the TCP protocol in the VM. In general, in SICC, the value of α_1 should be chosen to reflect the level of buffer occupancy that signals the drainage of the queue after incast; as such, in our simulations, we set the value of α_1 to 20% of the buffer size. In contrast, the value of α_2 should be chosen to signal the possible buffer overflow; hence, in the simulations and experiments, we set the value of α_2 to 100% of the buffer size. In addition, T should be set to a value larger than the end-to-end average RTT; consequently, if not otherwise stated, T was set to 1ms, or 10 times the RTT. DCTCP parameters were set according to the recommendations in [7], with $K = 15$ packets for 1Gbps links and $K = 65$ packets for 10Gbps links. We used the network simulator ns2 v2.35 [20], which we extended with the flow control receive window processing, to implement SICC. For DCTCP we used a patch available online for ns2 v2.35 from the authors [6]. For proper operation, the ECN-bit capability was enabled in the switch and TCP sender/receiver. Unless otherwise stated, the minRTO was set to 200ms (which is the default in Linux TCP implementations), the IP data packet size to 1500 Bytes, and the buffer size to 83 packets (or 125 KBytes).

5.1 Single-Rooted Tree Topology

First, we used a single-rooted (dumbbell) topology with 1Gb/s links and an average RTT of 100μs and ran the simulation for a period of 5 sec. Using mixes of 80 mice and elephant FTP flows, we simulated three scenarios, to mimic synchronized incast traffic competing with long-lived flows. In the first scenario, we simulated an elephant-to-mice ratio of 1:3 (i.e., 20 elephants and 60 mice) which is the ratio reported from private data centers in [7, 10]. In the second scenario, we increased the considered a ratio of 1:1. Then in the third scenario, we increased further the share of elephants to 3:1, in order to examine how SICC would respond when the network is highly loaded with long-lived (background) traffic. In all experiments, while the elephant flows sent at their allowed speed, without interruption, for the whole duration of the simulation, mice flows were set to have a finite supply of 10 KBytes data each. After finishing their transfers quickly, mice flows close their connections and reopen new ones at the beginning of each second. This resulted in a situation where the continuous, bulky, buffer-bloating elephants traffic is superimposed with 5 epochs of bursty incast traffic arrivals. To ensure that a relatively tight synchronization existed among mice flow arrivals in each incast epoch, we set the flows to start randomly within an average inter-arrival time of one packet transmission time.

For mice flows we report the CDFs of the Average FCT (AFCT), the cumulative packet drops experienced by mice flows, and the 99^{th}-percentile of the FCT for



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

Fig. 4: Performance of mice flows for TCP, SICC, RWNDQ and DCTCP in 1:3 ratio scenario.

mice flows only. Note that, the lower the values the better the performance. These results are shown in Fig. 4 for the 1:3 scenario, Fig. 5 for the 1:1 scenario and Fig. 6 for the 3:1 scenario. For elephant flows, we essentially report their goodput; and the results for the three scenarios are grouped in Fig. 7.

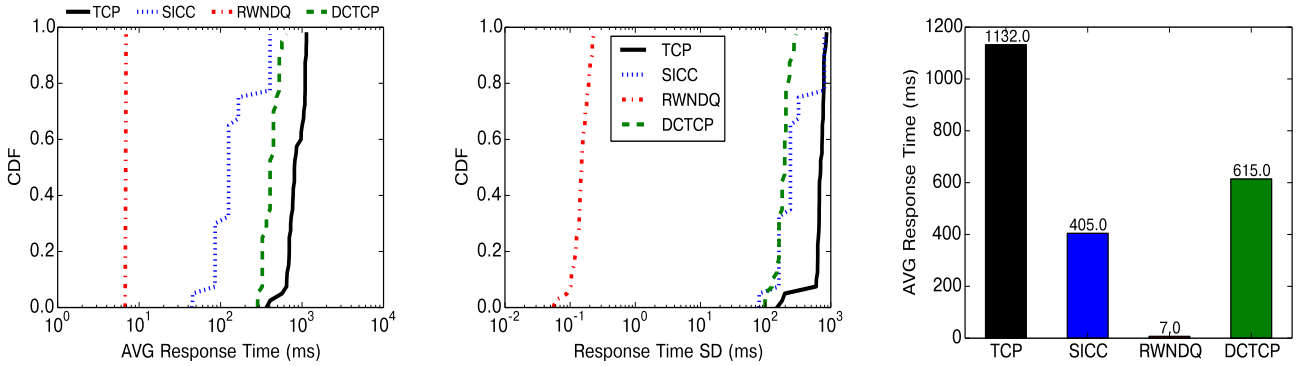
In the 1:3 scenario, Fig. 4a show that SICC can improve on average the flow completion time of mice flows compared to both TCP and DCTCP, and also (not shown here) the variance of the AFCT is improved compared to TCP and is in the same range as (slightly worse than) that achieved by DCTCP. This is expected as DCTCP's parameters are fine tuned for this particular ratio of mice to elephants. Being switch based, RWNDQ reduces further the AFCT and its variance due to its responsiveness and agility in setting the fair-share of the flows. Fig. 4b shows the total cumulative mice packet dropped at the bottleneck link during the 5 epochs. The figure gives the insight that SICC helps mice to achieve faster AFCT by reducing the frequency of packet drops, thus allowing TCP to avoid the huge penalty imposed by the minRTO. Finally, Fig. 4c shows the average (99th-percentile) of the completion time over the 5 epochs. Clearly SICC helps TCP to achieve a considerably faster FCT even on the tail giving an advantage for applications that generate co-flows without the need for a fully fledged complex scheduling mechanism as proposed in the literature (e.g., [12]).

In the 1:1 and the 3:1 scenarios, we replace the CDF of mice packet drops with the CDF of the standard deviation of mice FCT. Essentially, both Fig. 5a and Fig. 6a show similar improvements as before in terms of AFCT. Furthermore, Fig. 5b and Fig. 6b show that the SD of the FCT is better than that of TCP and is similar to or even better than that of DCTCP. Finally, as shown in Fig. 5c and Fig. 6c, the FCT improvement in SICC touches the great majority of mice flows, compared to DCTCP. this is due to SICC's ability to quench elephant flows temporarily during incast reducing thereby the frequency of mice packet drops. We notice in the highly loaded 3:1 scenario that DCTCP performs the worst (close to or even worse than TCP). Being switch-based, RWNDQ still gives the best performance.

Elephant Flows Performance: Fig. 7 clearly shows that in all three scenarios, SICC does not degrade the performance of elephant flows as it improves that of mice. SICC has nearly no impact on the achieved goodput compared to TCP. This can be attributed to the fact that SICC only intervenes temporarily during incast activity and its ability to restore immediately after the original sending rate of elephant flows.

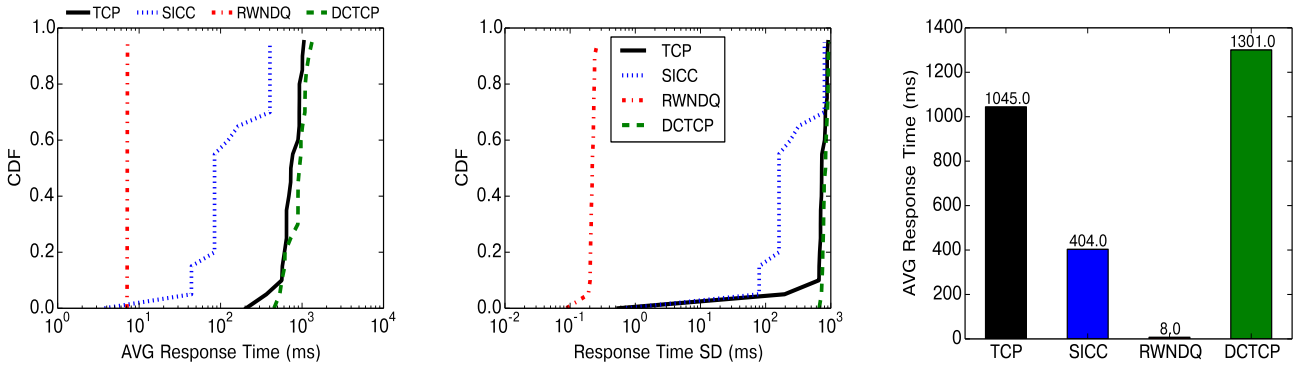
5.2 Fat-tree Datacenter Topology

To study SICC in a topology similar to those used in real data centers, we created a fat-tree like topology, as shown in Fig. 8, with 1 core, 2 aggregation, and 3 ToR switches. Each ToR switch connects to 48 servers with 1Gb/s link each. Each aggregation switch connects to each of the ToR switches with a 5Gb/s link and finally the core switch connects to each of the two aggregation switches with a 10Gb/s link. Such setup results in an over-subscription ratio of 1:24 at the ToR level (which is a moderate value by todays real DCs standards, where it is reported to reach up to 1:80). The one way propagation delay for each link is 25μs and the minRTO in the VM is still 200ms. In this scenario, the elephant traffic patterns are as follows: Rack1 to Rack3, Rack2 to Rack3 and Rack3 to Rack1. In contrast, to generate mice flows, we set the servers in Racks 1, 2 and 3 to host the worker tasks that send their results back to the aggregation server in rack 3, over 5 epochs during the simulation.



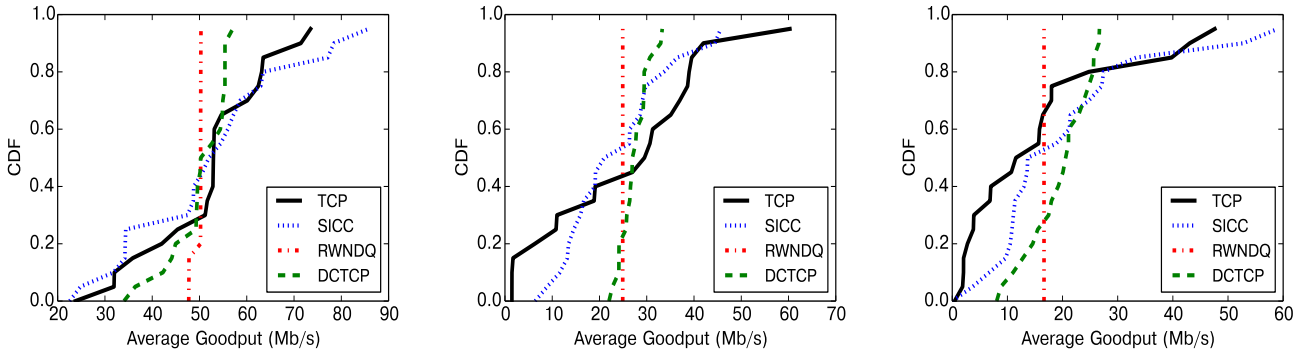
(a) Average FCT for mice (b) AFCT SD for mice (c) Average 99^{th}% FCT

Fig. 5: Performance 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. 6: Performance of mice flows for TCP, SICC, RWNDQ and DCTCP in 3:1 ratio scenario.



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

Fig. 7: Performance of elephant flows for TCP, SICC, RWNDQ and DCTCP.

Fig. 9 shows the results for this scenario. We can see clearly that SICC is able to improve incast flows FCT compared to TCP and DCTCP with nearly no impact on the elephants throughput CDF. As expected RWNDQ outperforms all schemes. The reduced FCT is mainly due to the reduced packet drops of short-lived mice flows.

We ran the simulation again, 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 were established from Rack(1,2) to Rack(3,4), Rack(3,4) to Rack(5,6) and Rack(5,6) to Rack(1,2). Fig. 10 shows the results. SICC and RWNDQ can improve TCP's performance and both achieve bet-

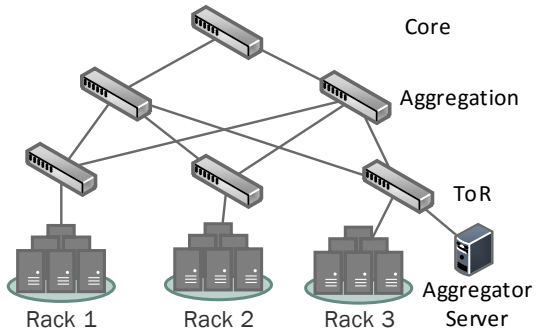


Fig. 8: A fat tree topology connecting 145 servers.

ter performance than DCTCP in a larger over-subscribed data center. The improvement is mainly due to the reduced mice packet drops and hence the average number of failed flows for SICC is reduced as shown in the legend of Fig. 10a.

5.3 Sensitivity of SICC to the monitoring interval

To study the sensitivity of SICC to the duration of the monitoring interval T, we repeated the simulation of the single-rooted topology with 1:3 elephant-to-mice ratio. We considered values of T equal to 1, 2, 10, 20, 25, 30, 50, and 100 times the RTT value of 100 μs.

Fig. 11 shows for mice flows, the CDF of the AFCT, the CDF of the SD of the FCT and the CDF of the 99th-percentile and for elephant flows, the CDF of average goodput. Fig. 11d 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. 11a, 11b and 11c 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 data centers, with a minimum RTT of 200-250μs, this translates to reading the queue occupancy once every 4-7ms which seems to be 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_1 and α_2 parameter, we found that SICC is not sensitive to these values. This is expected because the choice of the parameters α_1 and α_2 only affects at which point the incast flag is turned ON and OFF. And, since the buffer sizes of the switches in data centers are small, a slight change in these thresholds from the recommended settings adds only a fraction of a few microseconds to the time at which the incast flag is set to ON/OFF.

5.4 System Overhead

In terms of bandwidth overhead, we can simply 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 (with 48 ports per switch) and 1 controller and assuming a probing interval of 5ms then the payload message of size $\approx \frac{48port \times 2bytes}{queuesize}$ plus the 54 bytes for TCP, IP and Ethernet headers yields a 150 bytes message per switch. In total, for the 1000 switches the controller would receive 150 KBytes every 5ms, which translates into a bandwidth 240 Mbit/s. We believe this is reasonable bandwidth utilization for the communication overhead between the switches and the controller with respect to the performance gain for the majority of incast flows in data centers. In addition, in most current SDN setups, control plane signaling is out-of-band [24].

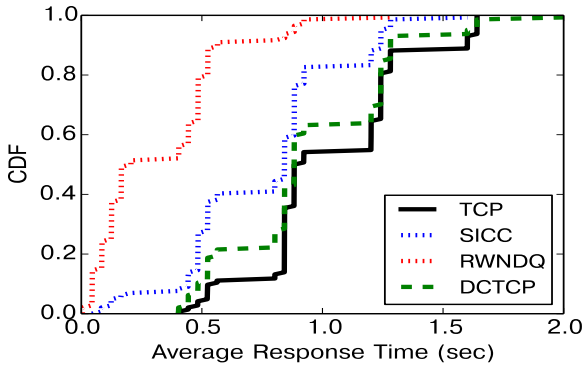
6 Testbed implementation of SICC framework

We further investigated the implementation of SICC as an application program integrated with the Ryu SDN controller [26], 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 data-path modules of Open vSwitch (OvS) [23] with the window update functions described in Section 4.2. We added the update function in the processing pipeline of the packets that pass through the data-path of OvS[2]. 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.

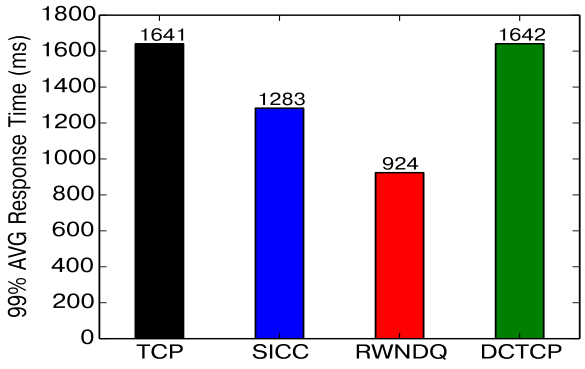
6.1 Testbed Setup

For experimenting with our SICC framework, we set up a testbed as shown in Fig. 12. All machines' internal and the outgoing physical ports were connected to the patched OvS on the end-hosts. We have 4 racks where rack 1, 2 and 3 were assigned the sender role and

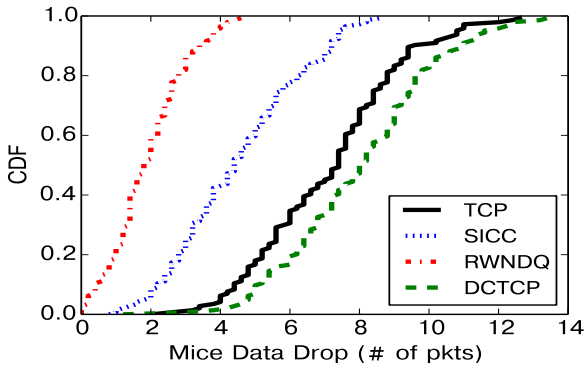
[2] Typical the throughput of internal networking stack is 50-100Gb/s. This is fast enough to handle tens of concurrent VMs sharing a single or several physical links. Hence, the window update function added to the vswitch would not hog the CPU nor affect the achievable throughput.



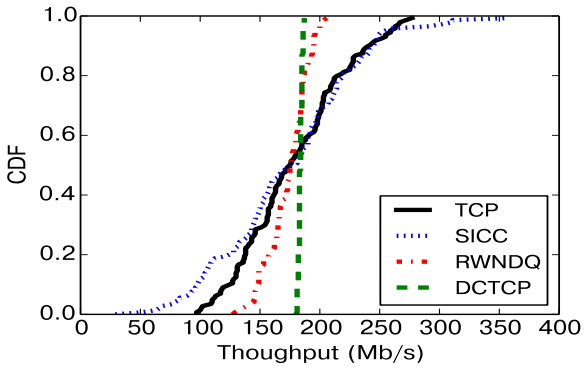
(a) Average FCT for mice



(b) Average 99^{th}% FCT

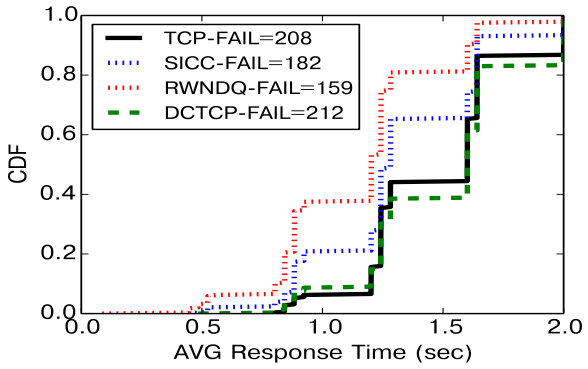


(c) Mice packet drop

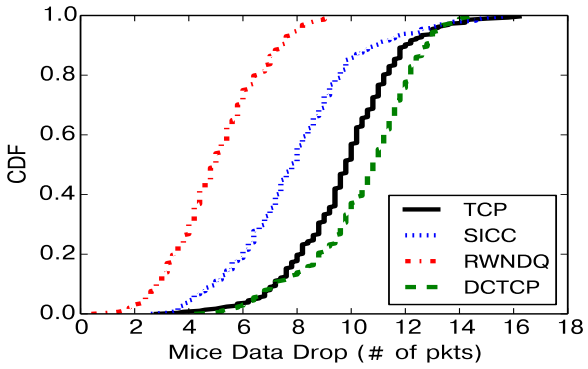


(d) AVG elephant goodput

Fig. 9: Performance of TCP, SICC, RWNDQ and DCTCP in small fat-tree topology of 144 servers.



(a) Average FCT for mice

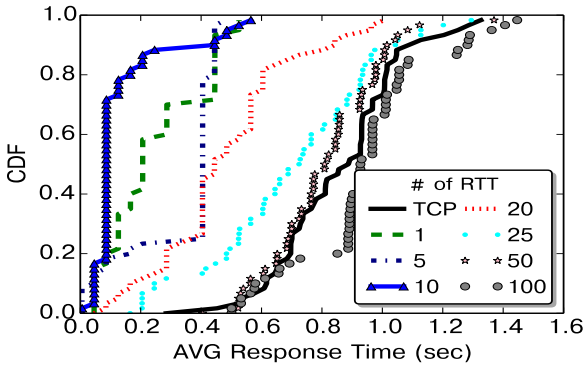


(b) AVG mice data drop

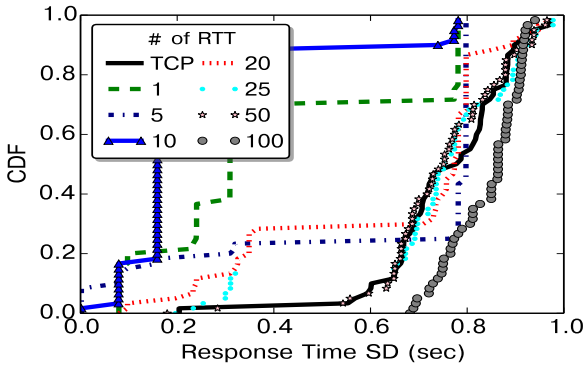
Fig. 10: Performance of TCP, SICC, RWNDQ and DCTCP in larger fat-tree topology of 288 servers.

rack 4 was assigned the receiver role. Each rack has 14 servers, however we ran the experiments on 7 of them only. The servers are installed with Ubuntu Server 14.04 LTS running kernel version (3.16) and are connected to the ToR switch through 1Gb/s links. The core switch in the testbed is a software OvS switch running on another server (i.e., the 8^{th} server in one of the racks). The servers are equipped each with a quad-port NIC which we used for the experiments and another dual-port NIC used for signaling and management. The OvS was setup to enable field matching on the TCP flags [22][3]. Similarly, the VMs were installed with the iperf program [14], to generate elephant flows, and the Apache web server hosting a single **"index.html"** webpage of size

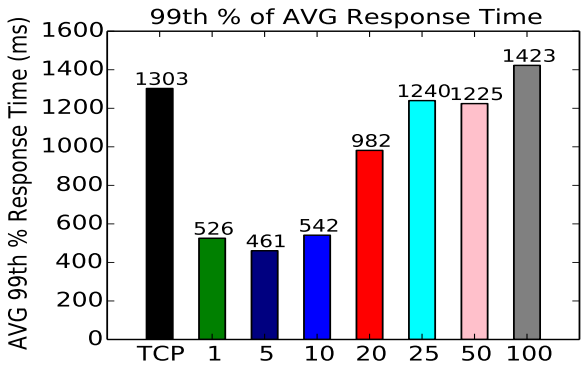
[3] The hardware switch was not used because its OF-DPA implementation does not follow OF1.5 specifications [22] which allows for matching on TCP flags. The support for OF1.5 is introduced into OvS starting from version 2.3.



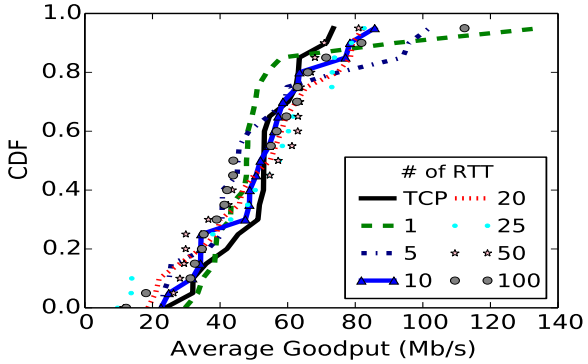
(a) Average FCT for mice



(b) AFCT SD for mice



(c) Average 99th % FCT



(d) AVG elephant goodput

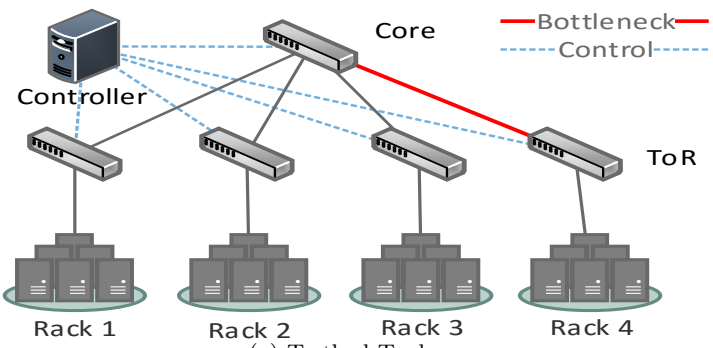
Fig. 11: SICC with variable queue monitoring interval.

11.5 KByte, to generate mice flows. We setup different scenarios to reproduce both incast and buffer-bloating situations. The bottleneck link in the network is shown in Fig. 12. The sending processes were 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 allowed us to create scenarios with a large number of flows in the network. In the testbed, the base RTT ranged from ≈200-300μs without queuing and up to 1ms with excessive queuing, hence we set the controller monitoring/sampling interval to a relatively large values in ≈10-50ms.

6.2 Experimental Results

The goals of the testbed 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 to overcome incast congestion situations in the network by improving mice completion time; *iii)* Study SICC's impact on the achieved throughput of elephants.

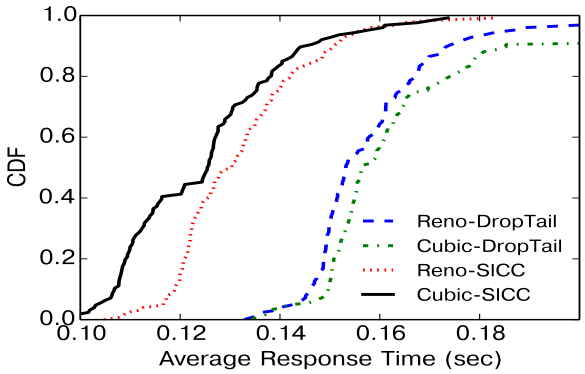
In the first experiment, we produced a scenario with incast and buffer-bloating using TCP NewReno. In this scenario, we generated 7 synchronized iperf elephant connections from each sender rack, to continuously send data for 30s, resulting in $7 \times 3 = 21$ elephants sharing the bottleneck link. Then, halfway through the experiment at the 15^{th} second, using Apache benchmark [9], we requested **"index.html"** webpage from each of the 7 web servers at each of the sending racks resulting in $7 \times 6 \times 3 = 126$ mice flows running on the same machines as the iperf servers. Each of the Apache benchmark processes requested the webpage 10 times before it reported different statistics over the 10 requests. We also repeated the previous experiment but in this case using TCP cubic as the congestion control. Fig. 13 shows that, in both cases, SICC achieves a good balance in meeting the conflicting requirements of elephants and mice. Specifically, Fig. 13d 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. 13d and Fig. 13b. In addition, as SICC efficiently detects the incast and proactively throttles the elephants, it can



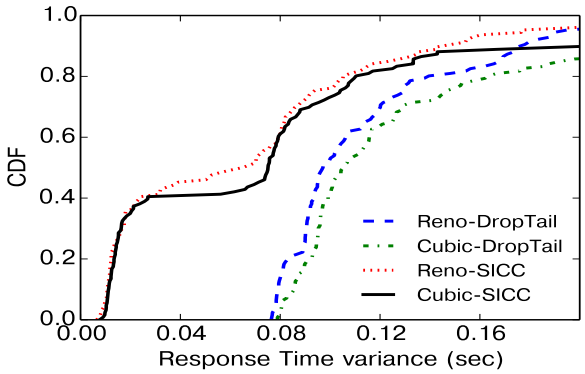
(a) Testbed Toplogy

(b) The actual testbed

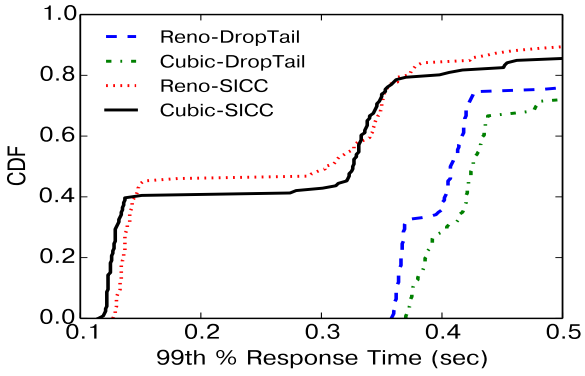
Fig. 12: A real SDN testbed for experimenting with SICC framework using Ryu-Controller, OpenFlow switches and OpenvSwitch



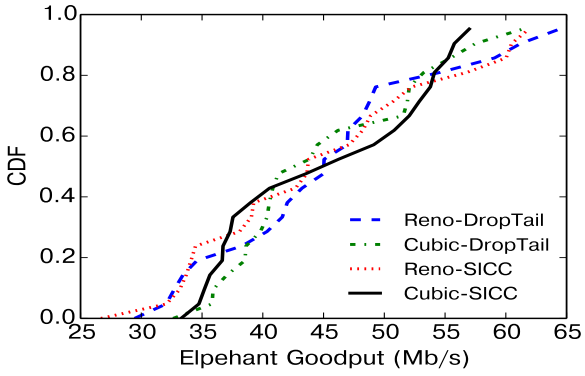
(a) Average FCT for mice



(b) SD of FCT for mice



(c) 99th % of FCT for mice



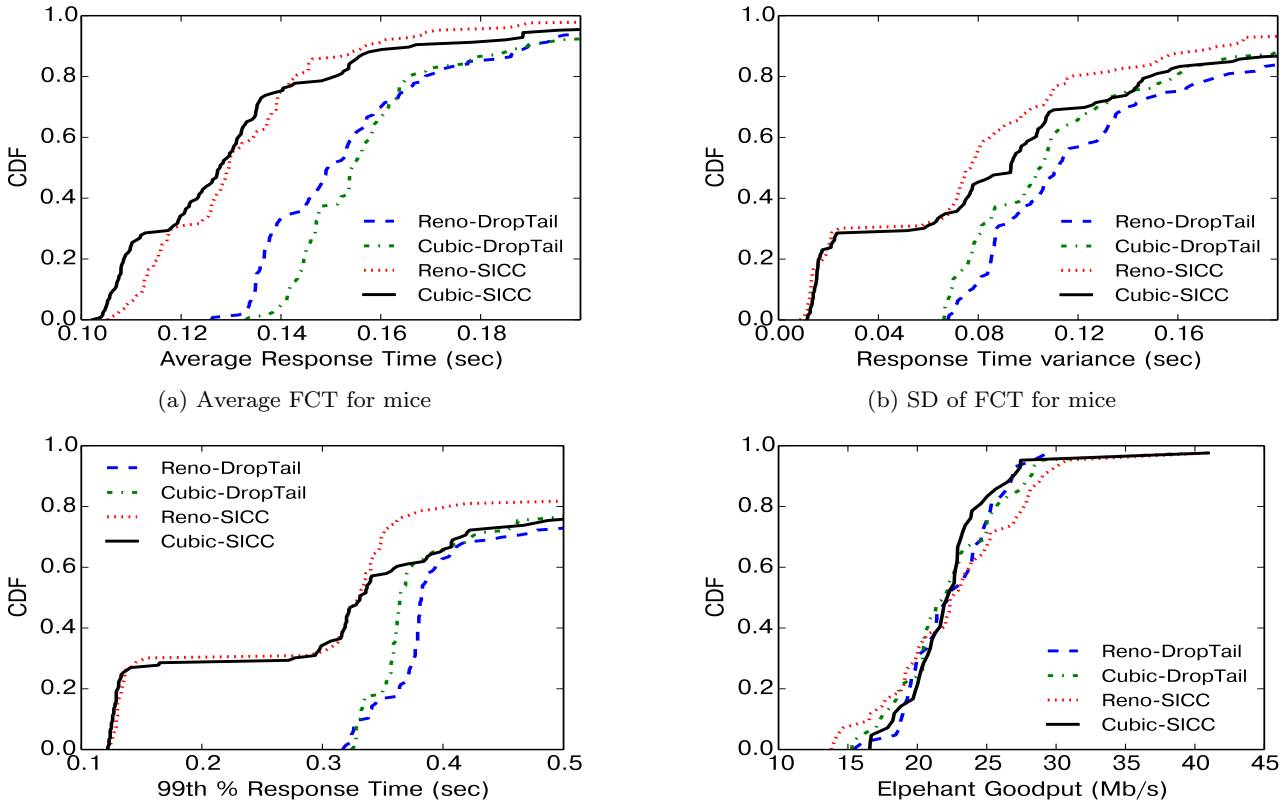
(d) AVG elephant goodput

Fig. 13: TCP-SICC vs. TCP-Droptail: 126 mice incast-like flows competing with 21 long-lived elephants

decrease the flow completion time even on the tail (i.e., 99th percentile) as shown in Fig. 13c.

We repeated the experiment by doubling the number of iperf flows per sender leading to $7 \times 3 \times 2 = 42$ elephants on the bottleneck. Fig. 14 shows that SICC still achieves a reasonable performance improvement for both TCP NewReno and TCP Cubic. Fig. 14d shows that long-lived elephant flows are not affected by SICC. Fig. 14a shows that mice flows still benefit under SICC even in a situation where buffers are heavily under pressured by the large number of elephants.

Even though, the experimental results show performance gains, they do not show the same gains as the simulations. This is because the simulation code in ns2 assumes ideal and predictable system behavior unlike the real system deployments. In the real system many uncontrolled factors may contribute to the system performance. For instance, we have used the soft-



(a) Average FCT for mice

(b) SD of FCT for mice

(c) 99th % of FCT for mice

(d) AVG elephant goodput

Fig. 14: TCP-SICC vs. TCP-DropTail: 126 mice incast-like flows competing with 42 long-lived elephants

ware version of the SDN-switches (i.e., Open vSwitch) not the hardware ones. In addition, the control network connecting the controller, the end-hosts and the switches uses old non-managed 100Mb/s switches while the data-path uses a DC-grade 1Gb/s switched network. These differences in the control and data path speed do not exist in the simulation. Finally, the Linux implementation of TCP contains many tweaks and added configurations that do not exist in the ns2 implementation. Despite the performance gains differences, the experimental results show that:

1. SICC helps in reducing mice traffic latency and maintaining a high throughput for elephants.
2. SICC handles incast events in low and high load scenarios without degrading the communication links utilization.
3. SICC achieves all of this without the need for any TCP stack and switching devices alternation.

7 Conclusion and future work

In this paper, we proposed an 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 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. SICC is also shown to be simple, practical, and easily deployable, meeting all its design requirements. Last but not least, 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 replacing the in-place commodity hardware switches. In this spirit, SICC's main contribution is to adhere to such principle while achieving great performance improvements. 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, FSGRF13EG14, REC14EG03 and FSGRF14EG24.

References

1. Abdelmoniem, A.M., Bensaou, B.: Efficient switch-assisted congestion control for data centers: an implementation and evaluation. In: Proceedings of the IEEE International Performance Computing and Communications Conference (IPCCC) (2015)
2. Abdelmoniem, A.M., Bensaou, B.: Incast-Aware Switch-Assisted TCP congestion control for data centers. In: Proceedings of the IEEE Global Communications Conference (GlobeCom) (2015)
3. Abdelmoniem, A.M., Bensaou, B.: Reconciling mice and elephants in data center networks. In: Proceedings of the IEEE CloudNet Conference (2015)
4. Abdelmoniem, A.M., Bensaou, B., Abu, A.J.: SICC: SDN-based Incast Congestion Control for Data Centers. In:Proceedings of the IEEE International Conference on Communications (ICC) (2017)
5. Akyildiz, I.F., Lee, A., Wang, P., Luo, M., Chou, W.: A Roadmap for Traffic Engineering in SDN-OpenFlow Networks. Computer Networks **71**, p.1–30 (2014).
6. Alizadeh, M.: Data Center TCP (DCTCP). http://simula.stanford.edu/ alizade/Site/DCTCP.html
7. 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**, p.63–74 (2010)
8. Alizadeh, M., Javanmard, A., Prabhakar, B.: Analysis of DCTCP: stability, convergence, and fairness. In: Proceedings of the ACM SIGMETRICS (2011)
9. Apache.org: Apache HTTP server benchmarking tool. Http://httpd.apache.org/docs/2.2/programs/ab.html
10. Benson, T., Akella, A., Maltz, D.a.: Network traffic characteristics of data centers in the wild. In: Proceedings of the ACM SIGCOMM (2010).
11. Chen, W., Ren, F., Xie, J., Lin, C., Yin, K., Baker, F.: Comprehensive understanding of TCP Incast problem. In: Proceedings of the IEEE INFOCOM (2015)
12. Chowdhury, M., Zhong, Y., Stoica, I.: Efficient coflow scheduling with varys. In: Proceedings of the ACM SIGCOMM, pp. 443–454 (2014).
13. Feamster, N., Rexford, J., Zegura, E.: The Road to SDN. Queue **11**, 20–40 (2013)
14. iperf: The TCP/UDP Bandwidth Measurement Tool. Https://iperf.fr/
15. Jouet, S., Perkins, C., Pezaros, D.: OTCP: SDN-managed Congestion Control for Data Center Networks. In: Proceedings of IEEE/IFIP Network Operations and Management Symposium (NOMS) (2016)
16. Kandula, S., Sengupta, S., Greenberg, A., Patel, P., Chaiken, R.: The nature of data center traffic. In: Proceedings of the ACM IMC (2009).
17. Karakus, M., Durresi, A.: A Survey: Control Plane Scalability Issues and Approaches in Software-Defined Networking (SDN). Computer Networks **112**, p.279–293 (2017)
18. Lu, Y., Zhu, S.: SDN-based TCP Congestion Control in Data Center Networks. In: Proceedings of IEEE IPCCC (2015)
19. Mckeown, N., Anderson, T., Peterson, L., Rexford, J., Shenker, S., Louis, S.: OpenFlow : Enabling Innovation in Campus Networks. ACM SIGCOMM CCR **38**, p.69–74 (2008)
20. NS2: The network simulator ns-2 project. Http://www.isi.edu/nsnam/ns
21. Open Networking Foundation: SDN Architecture Overview. Tech. rep., Open Networking Foundation (2013)
22. opennetworking.org: OpenFlow v1.5 Specification. Https://www.opennetworking.org/sdn-resources/openflow
23. openvswitch.org: Open Virtual Switch project. Http://openvswitch.org/
24. Panda, A., Scott, C., Ghodsi, A., Koponen, T., Shenker, S.: CAP for networks. In: Proceedings of the ACM HotSDN workshop (2013)
25. Rijsinghani, A.: RFC 1624 - Computation of the Internet Checksum via Incremental Update (1994). Https://tools.ietf.org/html/rfc1624
26. Ryu Framework Community: Ryu: a component-based software defined networking controller. Http://osrg.github.io/ryu/
27. 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**, p.303–314 (2009).
28. Wu, H., Feng, Z., Guo, C., Zhang, Y.: ICTCP: Incast congestion control for TCP in data-center networks. IEEE/ACM Transactions on Networking **21**, p.345–358 (2013)
29. Zhang, J., Ren, F., Tang, L., Lin, C.: Modeling and Solving TCP Incast Problem in Data Center Networks. IEEE Transactions on Parallel and Distributed Systems **26**(2), p.478–491 (2015)