SDN-based Incast Congestion Control Framework for Data Centers: Implementation and Evaluation

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

October 14, 2016

Abstract

Due to the partition/aggregate nature of many distributed cloud-based applications, incast traffic carried by TCP abounds in data center networks (DCNs). TCP, being agnostic to such applications' traffic patterns and their delay-sensitivity, cannot cope with the resulting congestion events, leading to severe performance degradation. The co-existence of such incast traffic with other throughput-hogging elastic traffic flows in the network worsens the performance further. In this paper, relying on the programmability of Software Defined Networks (SDN), we address this problem in an efficient and easily deployable manner to make it appealing to DCN operators and clients alike. We propose a SDN-based incast congestion control framework based on SDN controller and the hypervisor/vswitch programmability without modifying the guest VM nor the SDN-capable switches. We assess the performance of the proposed scheme via real deployment in a small-scale testbed and ns2 simulation in larger environments. The results demonstrate that the proposed framework can help incast traffic achieve faster completion time without affecting long-lived flows performance.

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 (we call these in the sequel mice); to long-lived, time-insensitive, bandwidth-inclined flows, such as those resulting from backups and virtual machine migration (we refer to these in the sequel as elephants). In particular, recent studies [11, 23] have shown that while in practice DCNs are crowded with mice, the lion's share in volume of traffic still goes to the elephants. Furthermore, many mice applications typically generate incast traffic. Typical applications that fall in this category include: *i)* **large-scale web applications** where every requested service is broken into parallel tasks assigned to worker nodes. The responses from these workers are collected by an aggregation node that finally produces the final result; *ii)* **distributed file storage** in which huge amount of data are stored in a number of distributed storage nodes, such as BigTable, HDFS and GFS. In this case, when a client recalls data, parallel access to some of these distributed nodes is needed; and, *iii)* **data processing applications** used mainly by many web-search, e-commerce, and social networks applications. Such systems, as an example MapReduce [8], can process huge amounts of data by partitioning and processing them in parallel across many servers. Hence, at the end of processing, synchronized many-to-many or many-to-one data transfers take place between the involved nodes.

DCNs are structured to provide a high bandwidth and low latency networking environment. To this end, and for cost considerations, Ethernet switches with small buffers (instead of routers with large buffers) are used for interconnecting the servers. In the presence of such small buffers, the sudden surge of synchronized incast traffic results in congestion events which are exacerbated by the presence of elephant traffic in the same buffer. Such complex congestion events are shown in recent works [5, 23] to be inadequately handled by TCP, as it is agnostic to the latency requirements of mice traffic flows as well as to the composite nature of the application data. Yet most applications in DCN still rely on TCP for data transport.

Recent works [3, 21, 1, 5, 23, 12, 2] tried to address this issue through different approaches. [5] proposed the well-known DCTCP which adopts TCP-AQM as a means to controlling congestion problems in DCNs. DCTCP modifies TCP congestion window adjustment function to maintain a high bandwidth utilization and sets RED's parameters to a small threshold to achieve a small queue length (and thus a short queuing delay). It is shown in [5] that DCTCP can achieve small delays for mice traffic without degrading the link utilization. Nevertheless, DCTCP requires the modification of TCP sender and receiver algorithms as well as fine tuning of RED parameters at the switches.

RWNDQ [3, 1] was proposed as an efficient TCP switch-assisted queue management mechanism in which the switch/router tracks the number of established flows across each individual queue, calculate a fair share for each flow that traverses it, and modifies the advertised receiver window to feedback this fair share to the sources. By leveraging the flow control apparatus which is an essential part of all TCP flavours in all operating systems (i.e., all VMs), RWNDQ manages to achieve better performance that the alternative approaches without modifying the TCP protocol code (which may not be possible in public data centers). Nevertheless RWNDQ requires the modification of the switch software making it hardly 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 to a dedicated central controller(s) with a global-view of the network. OpenFlow [13] is currently the dominant standard interface between the control and data path. This technique enables rich networking functions to be easily implemented and deployed on top of the SDN controller, yet, it has not been extensively explored as a potential framework for addressing incast congestion in datacenters.

1.1 Motivation and Objectives

We believe that any solution to the incast problem should be appealing to both the client and the cloud operator. Hence, we argue that modifying the TCP protocol and/or the hardware switching logic is only applicable 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. As a result, our target in this paper is to craft a solution to the incast problem that has the following requirements: (R1) it should handle effectively the problem of incast traffic congestion by improving the incast flow completion time; (R2) it should not degrade the throughput of elephant flows to achieve this; (R3) it should not modify the TCP sender nor receiver, and should not alter the hardware switches. If any modification is needed, it must be in the software of the programmable devices (i.e switch controllers and/or hypervisors/vswitches) that are fully under the control of the DCN operator; (R4) and last but not least it must be simple enough to be prone to deployment in a real system.

To this end, we adopt a SDN-based approach to congestion avoidance, where the data center controllers actively monitor the occurrence of incast traffic and proactively entice the sender's hypervisors to inhibit the TCP senders, whenever congestion events are foreseen to be imminent, to make room for mice traffic allowing them to pass with minimal congestion. When congestion recedes, the hypervisors entice the TCP senders to recovering their prior sending window immediately to sustain a high throughput. This is all done without modifying TCP.

In the remainder of this paper, we will first discuss our proposed methodology in Section 2 then present our SDN-based framework and discuss it in Section 3. We will first evaluate our framework via ns2 simulations in Section 4 to compare it to alternative approaches, then in Section 5 we discuss our implementation and evaluation in a small-scale testbed. We finally conclude the paper in 7.

2 Proposed Methodology

A simple illustration of the basic idea of our proposed SDN-based Incast Congestion Control (SICC) is given in Fig. 1. Assuming the persistent queue length in a SDN switch buffer at time round T_i to be $Q(T_i)$, if during T_i, a volley of N new TCP connections are established (i.e., N TCP *SYN* packets are seen) and no connection tear-down (i.e., no TCP *FIN* packets are seen), it is expected that after 1 RTT T_{i+1}, the queue length $Q(T_{i+1})$ is no more than $Q(T_i) + N * x * MSS$ bytes, where x is the initial window size of TCP. Should the queue length $Q(T_{i+1})$ reach a pre-set congestion threshold, knowing that incast traffic is ephemeral and that the persistent queue is mainly due to elephants, the ongoing and new flows are throttled to a sending rate of 1 MSS per RTT. This ultimately achieves a short term fairness among all flows (mice and elephants) during the lifetime of incast traffic, hence meeting requirement (R1).

In principle, since the TCP source rate is determined by the sending window *Swnd* which is the minimum of receiver window *Rwnd* , and the congestion *Cwnd* , and since *Cwnd* is normally at least equal to 1 MSS, to meet requirement (R3), the SDN controller, being aware of the possible incast event, can send a warning message to the sending end-hosts to start rewriting the receiver window field in the TCP ACK headers as a means to throttling the sender rate to 1 MSS per RTT. SDN also provides much useful statistics on the ongoing number of flows and the queue occupancy for each switch port. Notice that all the rewriting happens in the hypervisor below the virtual machines and does not interfere with the TCP protocol inside the VM.

Throttling all flows sending rates to a single segment per RTT will have the immediate effect of dropping the queue length dramatically below the congestion threshold, as a result, since incast traffic is ephemeral, to meet requirement (R2), *Rwnd*

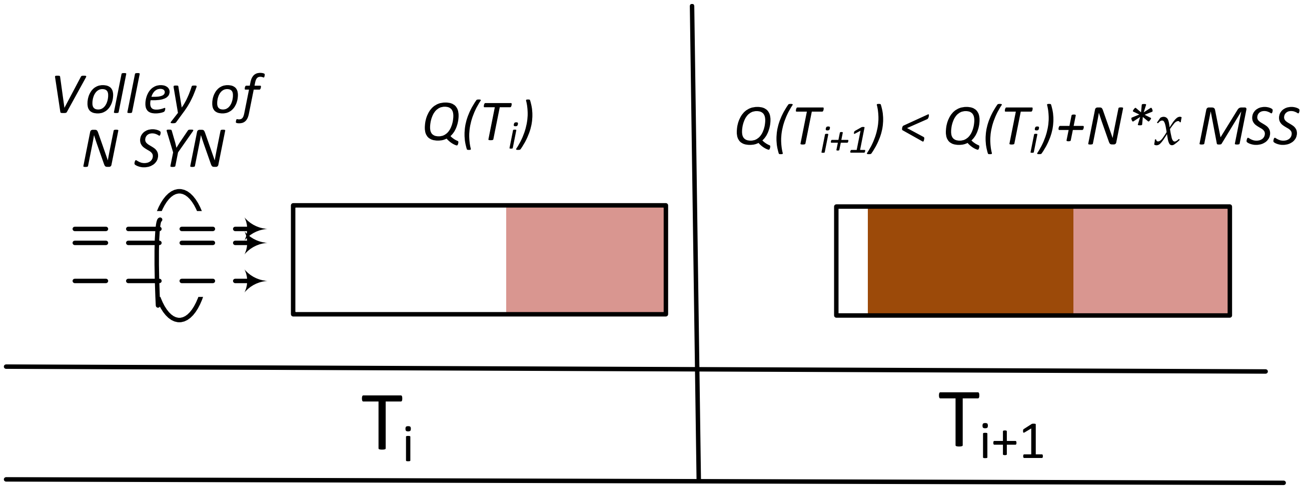


Figure 1: *SICC Idea Rationale*

rewriting would stop typically after a few time intervals, which enables ongoing elephants to recover their previous sending rate (since *Cwnd* and *Rwnd* are still the same). To meet requirement (R4), instead of tracking individual flow states to estimate accurately the queue length in the next interval, the controller can use rough estimates by simply tracking and counting the number N of segments with a SYN bit set less the number of segments with the FIN bit set; this in the worst case results in a conservative estimate of the expected 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 2 shows the detailed protocol interactions among the different modules residing on the controller, switches and end-hosts as follows: 1. The controller's monitoring module is responsible for tracking, accounting and extracting information (i.e., the window scaling option) from incoming SYN/FIN. 2. The controller's warning module is responsible for predicting incast events based on SYN/FIN arrival rates and for sending out incast ON/OFF special messages directed to the involved senders' VM addresses. 3. Upon receipt of incast ON message for a certain VM, the SICC hypervisor/vswitch module will start to intercept and modify the incoming ACKs for that VM until an incast OFF message is received later or the incast event timesout. 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 switches to forward a copy of any SYN/FIN packet through the OpenFlow protocol interface.

Figure 3 illustrates a possible deployment scenario of our proposed SICC framework in SDN based data centers. All switches in the DC are SDN-enabled, the controller controls all the switches in the DC and sets rules in the ingress/egress router as well as all Top of Rack (ToR) switches to intercept any TCP SYN. As a result, the controller is able to track TCP connections and to pin-down forward and reverse paths by setting new forwarding rules (or merging them in existing aggregates) in the DCN switches. By also intercepting the FIN segments in the ingress/egress router and ToR switches, the controller is also able to withdraw routing rules from the switches as necessary. Each of the end-host's hypervisor/vswitch is patched to receive and process the incast warning originating from the central controller to update the receiver window field in the ACK headers as they arrive into the sending end-host or guest VM.

3 SDN-based Incast Congestion Control

The main variables and parameters used in the SICC framework are described in Table 1. Notice that T_i, T_{Mice} and α are parameters of the algorithm that can be chosen by the DC administrator. As a rule of thumb, T_i should be larger than 1 RTT.

3.1 SICCQ: Incast detection via Queue-based Monitoring

The central controller sets OpenFlow rule at all datacenter OpenFlow-based switches to forward a copy of any *SYN* or *FIN* packets to the controller. In most cases, TCP *SYN* packets contains optional TCP headers with useful information (i.e., maximum segment size and window scaling value), which is stored in source-destination hash tables to be used in our algorithm. Also, the controller inquiries for the port statistics over a fixed intervals to calculate a smooth weighted moving average of the queue occupancy. By doing so, the controller is able to detect possible congestion events using the following algorithm:

Table 1: *Variables and Parameters used in SICC framework*

Parameter name	Description
T_i	Timeout value for monitoring interval
α	Queue threshold to turn off Incast
T_{Mice}	Average time for mice to finish

Variable name	Description
β	Coarsely estimated differential of new connections
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
P	a packet
$Rwnd(P)$	Receiver window field in packet P
$Incast$	Boolean true if incast is ON
$IncastON$	Time at which incast set to ON
$SWITCH$	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

Algorithm 1 SICCQ Controller Algorithm

```
procedure $Packet\_Arrival(P, src, dst)$
    if $SYN\_bit\_set(P)$ then
        $\beta \leftarrow \beta + 1$
        $M[src][dst] \leftarrow P.tcpoption.mss$
        $W[src][dst] \leftarrow P.tcpoption.wndscale$
    if $FIN\_bit\_set(P)$ then
        $\beta \leftarrow MAX(0, \beta - 1)$
procedure $Incast\_Detection\_Timeout$
    for each $sw\ in\ SWITCH$ do
        for each $p\ in\ SWITCH\_PORT$ do
            $Q[sw][p] \leftarrow \frac{Q[sw][p]}{4} + \frac{3 \times q[sw][p]}{4}$
            if $now - IncastON[sw][p] >= T_{Mice}$ then
                if $q[sw][p] < \alpha \times B$ then
                    for each $dst\ in\ PORT\_DST$ do
                        for each $src\ in\ DST\_SRC$ do
                            $msg \leftarrow$ “$INCAST\ OFF\ to\ dst$”
                            $send\ msg\ to\ src$
            $Q\_next \leftarrow \beta \times Initial\_CWND + Q[sw][p]$
            if $\beta > 0\ and\ Q\_next \geq B$ then
                $IncastON[sw][p] \leftarrow now$
                for each $dst\ in\ PORT\_DST$ do
                    for each $src\ in\ DST\_SRC$ do
                        $msg \leftarrow$ “$INCAST\ ON\ to\ dst$”
                        $msg \leftarrow msg + W[src][dst]$
                        $msg \leftarrow msg + M[src][dst]$
                        $send\ msg\ to\ src$
    $\beta \leftarrow 0$; Restart Incast detection timer $T_i$
```

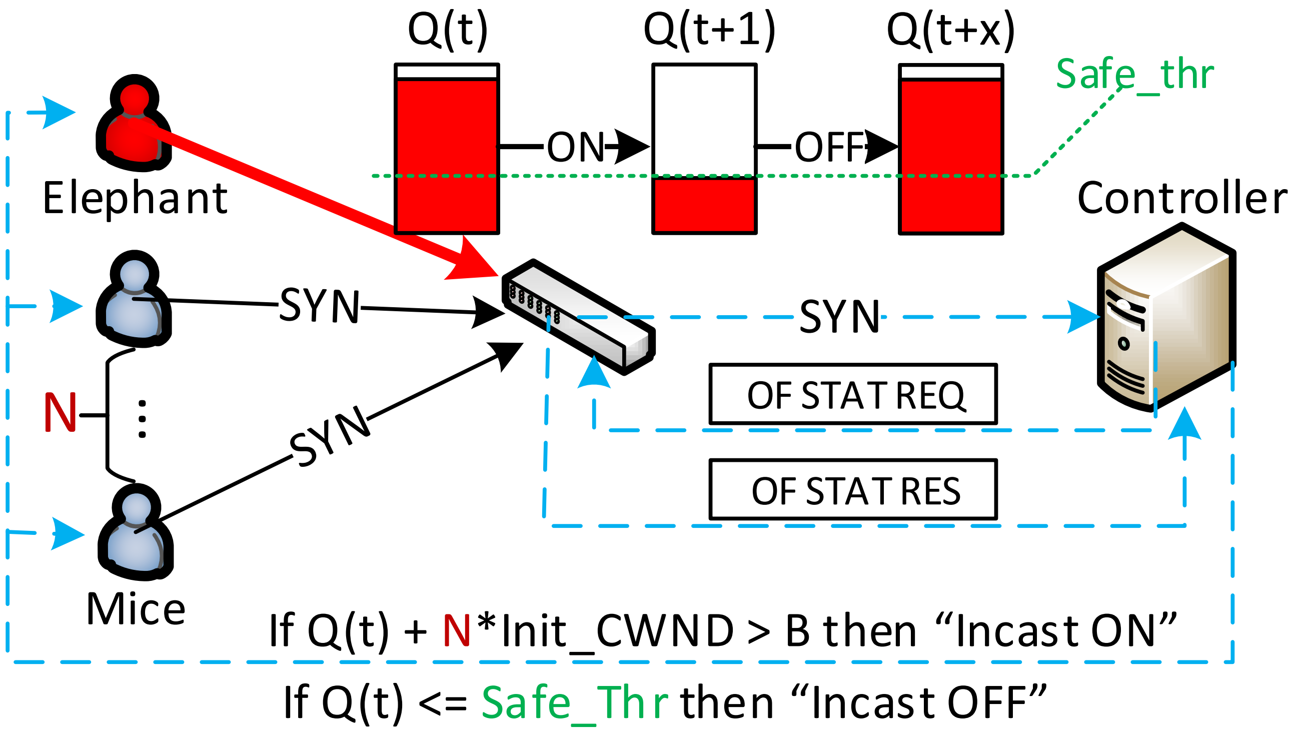


Figure 2: *A detailed view of SICC framework components' interactions which forms a closed-control cycle.*

SICCQ shown in Algorithm 1 is an event-driven mechanism which implements two major event handling: packet arrivals and incast detection timer expiry to trigger incast on or off messages to the involved sources.

1. **Upon a packet arrival:** If this is a SYN packet for establishing a new TCP connection, then the current value of switch port's β is incremented and the necessary information of the source establishing the connection are extracted from the TCP headers (i.e., the window scaling and the maximum segment size). Otherwise, If this is a FIN packet for closing an established TCP connection, then the current value of switch port's β 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 $Init\_Cwnd$ 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 we need to take a fast proactive action to make room for the forthcoming possible incast traffic. We immediately send to the hypervisor of senders involved in the incast situation a message to raise up their incast flag (INCAST-ON). On the other hand, if the buffer occupancy starts decreasing below the incast safe threshold (i.e., 20% of the buffer size) or the time since the incast ON is more than the transmission time of mice flows, then we send to the involved hypervisor in the incast a message to lower down their incast flag (INCAST-OFF).

3.2 Hypervisor Window Update Algorithm

The end-host's hypervisor or vswitch are to be patched and modified to track for any possible messages coming from the DC controllers and implement the action of resetting the receive window field to 1 MSS on the incoming ACK messages to guest VMs. Currently, for identifying controller messages, we use one of the unused (experimental) Ethernet types to indicate that the message is either incast ON or OFF messages. The hypervisor implements the following algorithm to act upon arrival of any 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 will turn ON the incast flag for this source-destination pair and extracts the attached information about the destination (i.e., the window scale shift factor and the maximum segment size) for ACK receiver window rewriting.

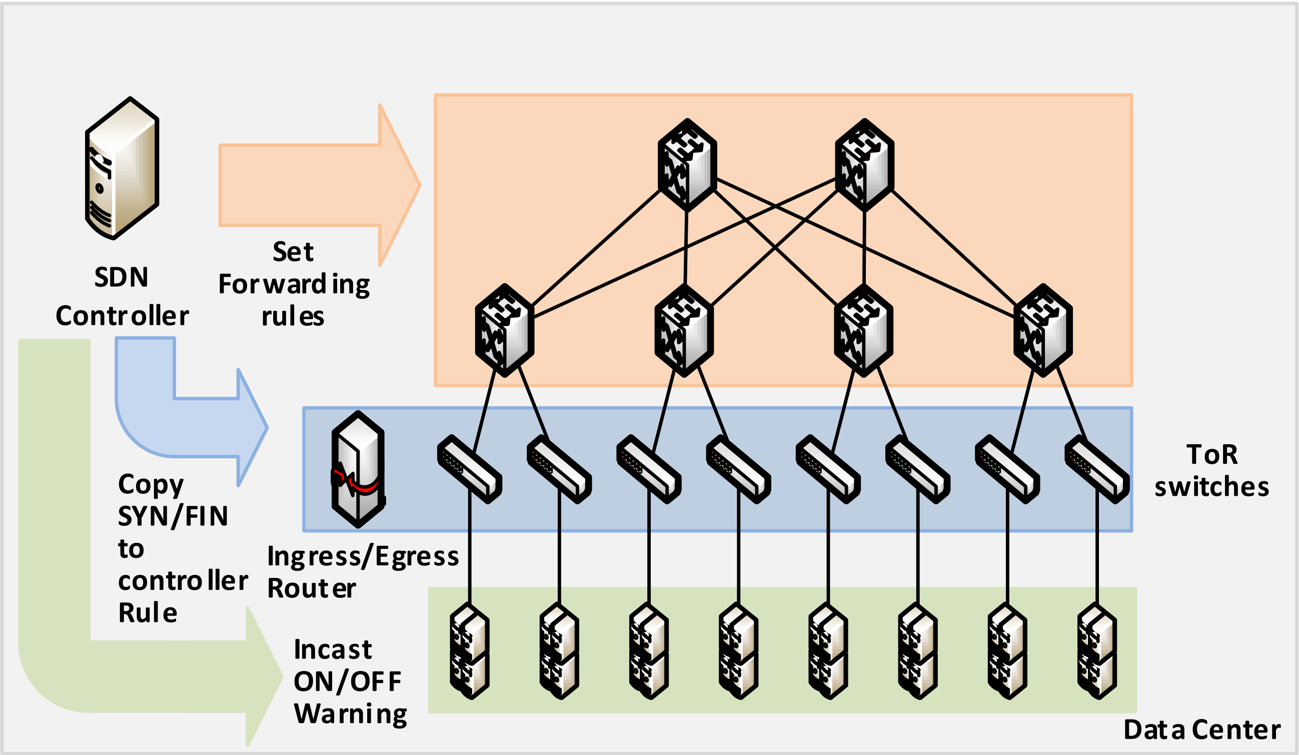


Figure 3: *A full SICC-based data center deployment with relations among end-hosts, switches and the controller.*

2. **Incast OFF:** If the received packet is identified as an "Incast OFF". Then the hypervisor turns OFF the incast flag (i.e., stop 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 is turned on for that source-destination pair, if so the hypervisor proceeds to update the receive 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 the 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 the 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 rate.

3.3 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 is appealing for handling the co-existence of mice and elephants. SICC adopts a proactive recovery actions in face of the forecast incast information. As soon as the incoming traffic gives indication of overflowing the buffer, the receive window is shrunk drastically to 1 MSS. Furthermore the new window is equally applied to all ongoing flows meaning that all flows (mice or elephants) will receive an equal treatment during incast periods. As soon as, the incast traffic, which is short-lived, finishes as is indicated by the queue occupancy dropping back to less than the predetermined incast-safety threshold, the elephants immediately restore their previous sending rates by disabling receive window rewriting.

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 are based on SDN. In addition, the window update mechanism at the hypervisor is so simple that it only requires $O(1)$ 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

Algorithm 2 SICC Hypervisor Algorithm

```
procedure $Packet\_Arrival(P, src, dst)$
    if $INCAST\_ON\_set(P)$ then
        $INCAST[src][dst] \ = \ true$
        $W[src][dst] \ = \ P.wndscale$
        $M[src][dst] \ = \ P.mss$
    if $INCAST\_OFF\_set(P)$ then
        $INCAST[src][dst] \ = \ false$
    if $ACK\_bit\_set(P)$ then
        $val \ = \ M[src][dst] >> M[src][dst]$
        if $INCAST[src][dst] \ and \ Rwnd(P) > val$ then
            $Rwnd(P) \ = \ newwnd$
            $Recalculate \ Internet \ Checksum \ for \ P$
```

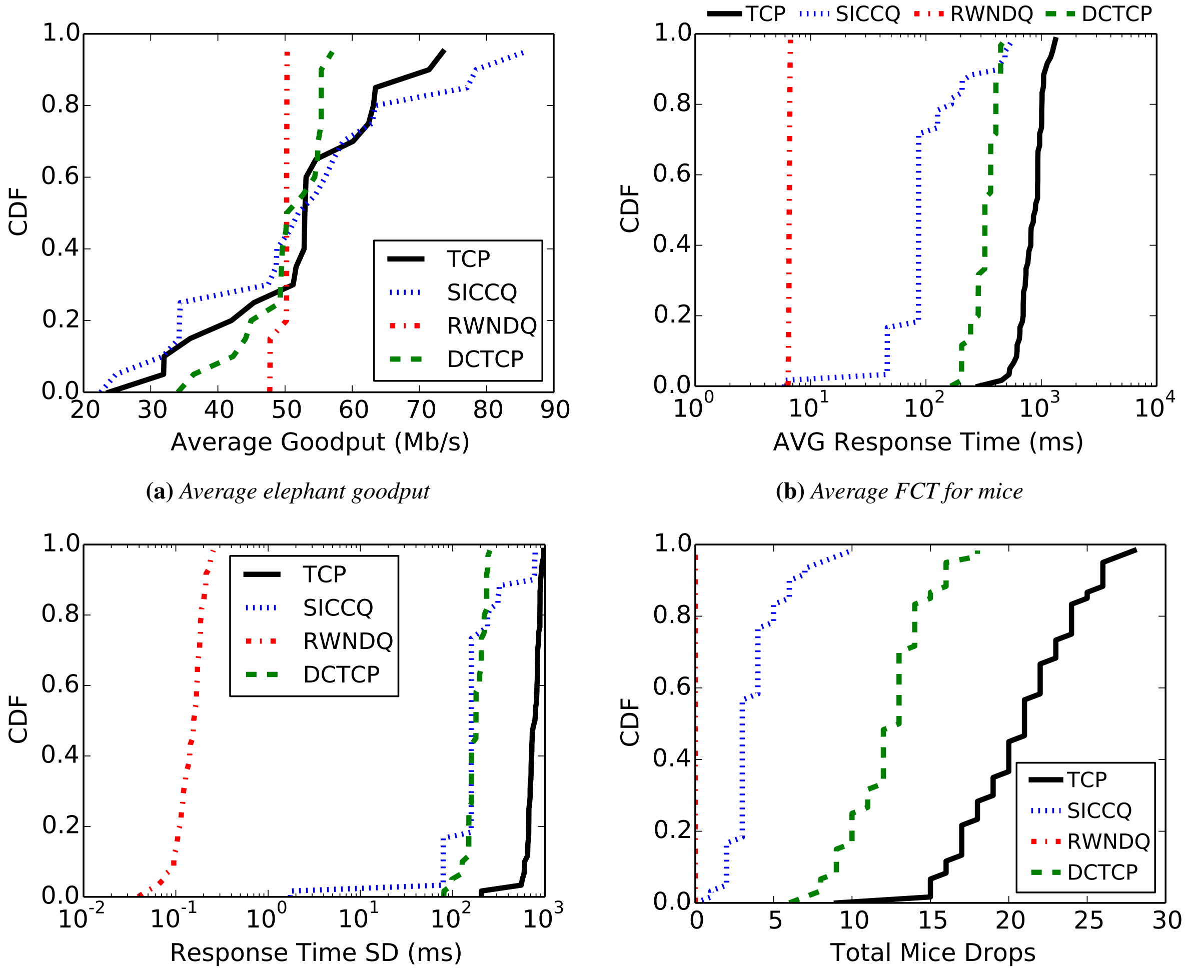
very easily and efficiently after header modification, by applying the following straightforward one's-complement add and subtract operations on three 16-bit words: $Checksum_{new} = Checksum_{old} + Rwnd_{new} - Rwnd_{old}$ [19]. This also takes $O(1)$ per modified packet. In addition, since SICC is 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 wild-card matching over all fields except for TCP flags which do not require per-flow information tracking, 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 the wide area Internet, such route changes are very highly unlikely to happen in DCs due to path stickiness and the reliance on switches.)

SICC framework may be susceptible to performance degradation when subjected to the famous SYN flooding attacks [9]. This attack normally exploits the flooding on one-way opened connections to exhaust server memory, however in our setup, it would force the receive window of the legitimate sources sharing the same queue to 1 MSS whenever a flood of half-open SYN are encountered. This attack may lead the senders' window to frequently fluctuating between the current full rate and 1 MSS per RTT. This well-known attack can affect the operation of any TCP flavor, DCTCP and ICTCP as well, because it is targeted towards TCP applications in general. Over the past few years, many proposals have suggested possible solutions to mitigate this attack [9]. One recent proposal that can secure SICC framework against SYN flooding attack is FloodGuard [22]. FloodGuard implements an efficient, lightweight and protocol-independent defence framework for SDN networks. It is shown that it is effective in mitigating flooding attacks while adding only negligible overhead into the SDN framework.

4 Simulation and Performance Analysis

In this section, we study the performance of our algorithm via simulation in network scenarios with a low delay high bandwidth (as is the case in data centers). We compare our system to TCP-DropTail, TCP-RED, RWND and DCTCP and demonstrate how it outperforms both TCP with Droptail or RED and achieves similar performance as DCTCP and RWNDQ yet requires neither modification to the TCP source and receiver nor to the switches. 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 be equal to the average round trip time in the network. In the simulation experiments, we set α to 20% of the buffer size, T_i to 10 μs (i.e., 10 times the RTT). DCTCP parameters are set according to the recommended settings by the authors with K (the target queue occupancy) set to 17% of the buffer size.

We use the network simulator ns2 version 2.35 [14], which we have extended with SICCQ framework. In addition, we modified ns2 TCP module, since the receiver window interaction between TCP sender and receiver (Flow Control) in ns2 does not follow the standard TCP flow control implementation of sending receive window values in the ACKs. We compare TCP NewReno over Droptail, SICCQ, RWNDQ and DCTCP (which involves a modification of TCP and AQM). For DCTCP, we use a patch for ns2.35 available from the authors [4] 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, a bottleneck link of 1 Gb/s, average RTT of 500μs and the MinRTO of 200ms which is the default in most linux TCP implementations.



(a) *Average elephant goodput*

(b) *Average FCT for mice*

(c) *AFCT SD for mice*

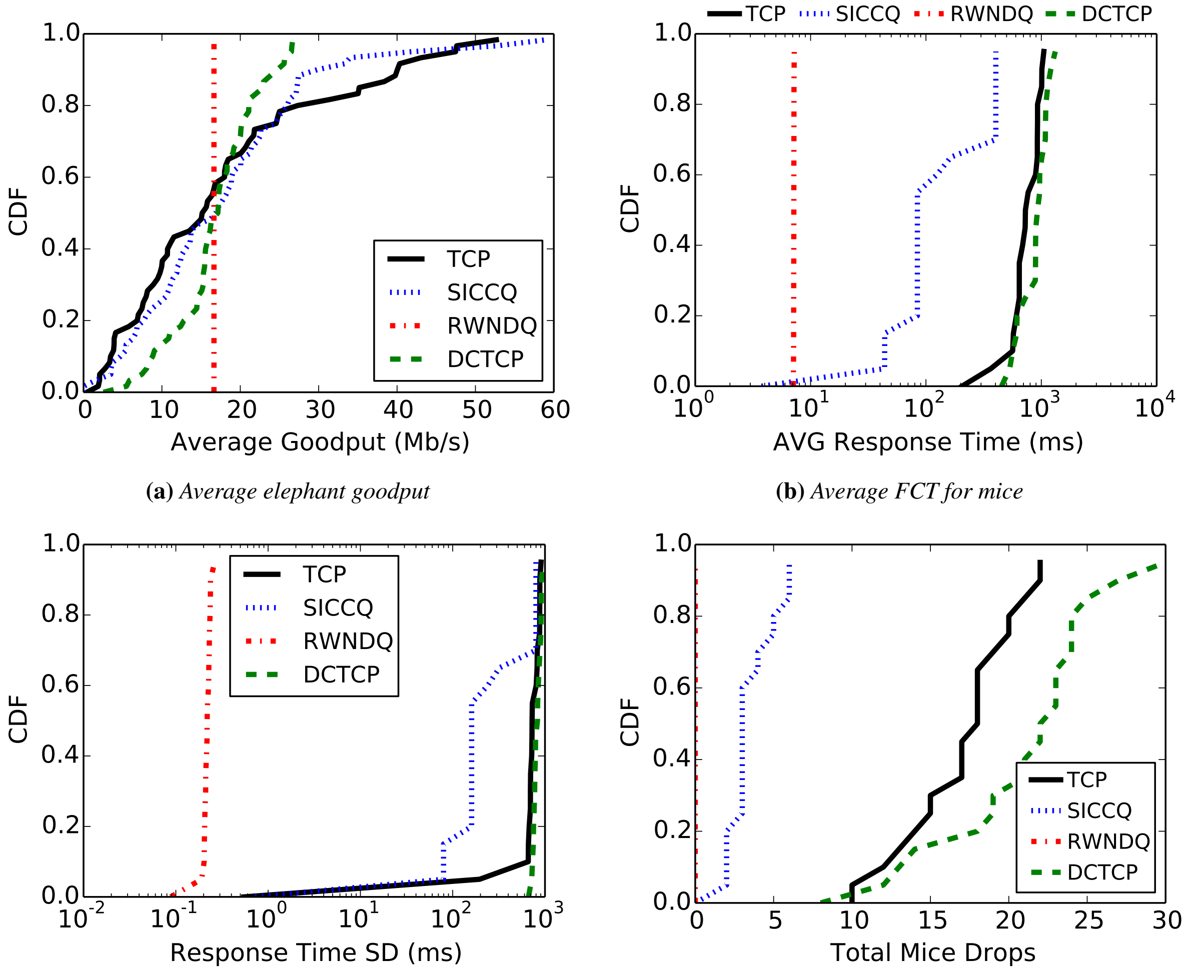
(d) *Mice packet drops*

Figure 4: *Performance metrics of TCP, SICCQ, RWNDQ and DCTCP in elephant-to-mice 1:3 ratio scenario.*

4.1 Single-rooted Topology Simulation

First, we use a single-rooted (dumbell) topology and run the experiments for a period of 5 sec. The buffer size of the bottleneck link is set in all cases to 83 packets or 125 KBytes where the mean IP data packet size is 1500 bytes. We simulate two scenarios to cause incast and queue-buildup situations at the same time where the number of sources are 80 FTP flows. 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 [5, 7]. We rerun the simulation but increase the share for elephants to examine how SICCQ would respond in situations where the network is highly loaded with long-lived (background) traffic. In this simulation, we set the ratio to 3:1. All sources start at same time at the beginning and while elephants keep sending at full link-rate during the whole simulation period, mice who finish their flow very quickly then close the connection reopen a new one at the beginning of each second (5 epochs during the whole simulation). To ensure a relatively tight synchronization between mice flows, and create an incast traffic scenario, in each of these 5 epochs, the individual mice start in a random order within one packet transmission time of each other. Each mouse sends 10KBytes of data then halts until the start of next epoch.

Fig. 4 shows the distributions of the mean and variance of the flow completion time (FCT) for mice, average (99th-percentile) completion time of mice and the average goodput for elephant flows in the lightly loaded 1:3 elephants to mice ratio scenario. Fig. 4a suggests that SICCQ has nearly no impact on the achieved goodput of TCP which means the elephants' performance is not degraded due to our scheme. Fig. 4b and 4c show that SICCQ can improve mice flow completion time on average with lower variation in response times, achieving a performance close to DCTCP. However, RWNDQ improves even further due to its agility in setting the fair-share of the flows as it is switch-based. SICCQ can improve TCP's performance yet requires no modification to the communication end-points nor the switches. Finally, Fig. 4d shows the total cumulative mice



(a) *Average elephant goodput*

(b) *Average FCT for mice*

(c) *AFCT SD for mice*

(d) *Mice packet drops*

Figure 5: *Performance metrics of TCP, SICCQ, RWNDQ and DCTCP in elephant-to-mice 3:1 ratio scenario.*

packets drops during the 5 epochs at the bottleneck link. This gives an insight on how SICCQ is helping mice to achieve faster FCT by reducing packet drops thus allowing TCP to avoid the huge penalty of waiting for timeout.

In the 3:1 elephants-to-mice ratio case, Fig. 5a suggests that SICCQ again has achieved the same goodput of TCP. Fig. 5b and 5c show that SICCQ can still improve the mice average FCT with low variation close to or better than DCTCP. RWNDQ still gives the best performance in this highly loaded case. Finally, Fig. 5d still shows that SICCQ is able to reduce TCP's drop probability at the bottleneck and hence reduces the FCT. The results suggests that the drops are worse when a synchronized burst of packets hit the queue, leading to a window-full of data to be lost or a case where fast (i.e., 3 dup-ACK) recovery is not possible.

4.2 Fat-tree Datacenter Topology Simulation

We have created a fat-tree like topology as show in Fig. 6 with 1 core, 2 aggregation, and 3 ToR switches each connecting 48 servers to mimic the topologies used in real data center environment. We connected an aggregation server at the 3rd rack which receives the result from all 144 server in the network. We use in our simulation experiments links of core-agg=10Gb/s, ToR-agg=5Gb/s and Server-ToR=1Gb/s links. This setup creates an over-subscription 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 propagation delays of 25μs per link and MinRTO of 200ms. In this scenario, the elephants communicates as follows: Rack1→Rack3, Rack2→ Rack3 and Rack3→Rack1. Mice communication defines Rack 1, 2 and 3 to be the workers who are sending results back to the aggregation server in 5 epochs during the simulation.

Fig. 7 shows the results for this scenario. SICCQ is able to improve incast flows FCT comapred to TCP and achieves comparable performance as DCTCP with nearly no impact on elephants throughput. As expected RWNDQ outperforms all

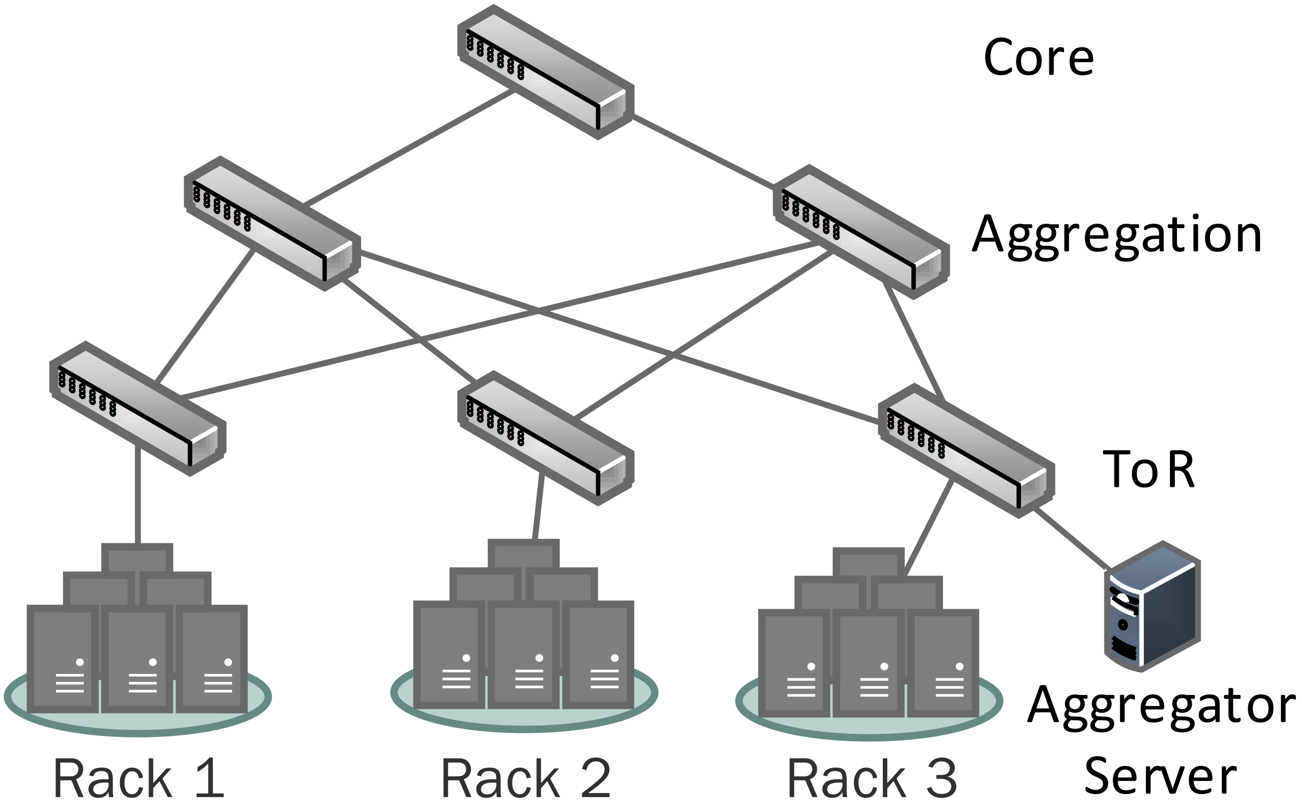


Figure 6: *A fat tree topology connecting 145 servers.*

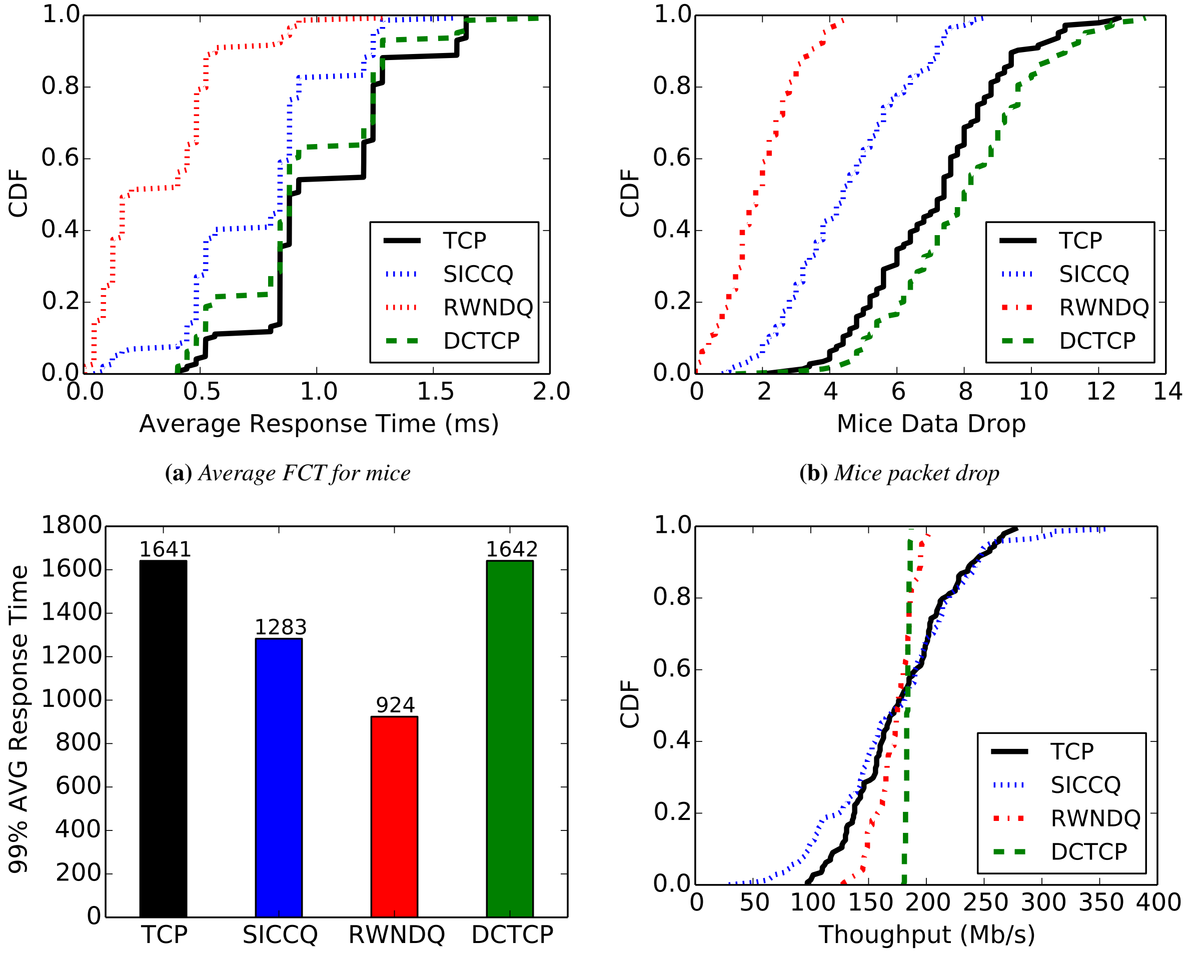
schemes due to its accurate estimation of the fair-share at the switch. The reduced FCT is mainly attributed to the reduced packet drops of short-lived mice flows.

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. Elephants flows are defined as Rack(1,2)$\rightarrow$Rack(3,4), Rack(3,4)$\rightarrow$Rack(5,6) and Rack(5,6)$\rightarrow$Rack(1,2). Fig. 8 shows the results. SICCQ and RWNDQ can improve TCP's performance and both achieve better performance than DCTCP in a larger but more relaxed 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 SICCQ are reduced as shown in Fig 8a's legend.

4.3 Sensitivity of SICCQ to the monitoring interval

We repeat the single-rooted experiment with 1:3 elephant-to-mice ratio for SICCQ 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 SICCQ Algorithm 1 normalized to the RTT value (i.e., 100 μs). To cover a wide range of values we simulated it for (1, 2, 10, 20, 25, 30, 50, 100) worth of RTTs. Fig. 9 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. 9a implies that SICCQ's monitoring interval does not affect the achieved goodput of TCP but it would affect the efficiency of SICCQ's incast detection ability. Fig. 9b, 9c and 9d show that SICCQ can still achieve a good performance, even with a monitoring interval 25 times wider than the RTT in the network. This analysis suggests that a value $\approx$ 1-25 RTT in the network would be sufficient[1]. This justifies the choice of a monitoring interval of 10 times the RTT in the previous simulations. We did a sensitivity analysis through multiple simulations (not shown here) on the value of α and $safe\_thr$ parameter, we found that SICCQ 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-queuesize(payload) +

[1] In our testbed, with a minimum RTT of 200-250μs, this translates to reading the queue occupancy once every 4-6.5ms. We believe this is an acceptable value for the controller to probe the switches



(a) *Average FCT for mice*

(b) *Mice packet drop*

(c) *Average 99th % FCT*

(d) *Average elephant goodput*

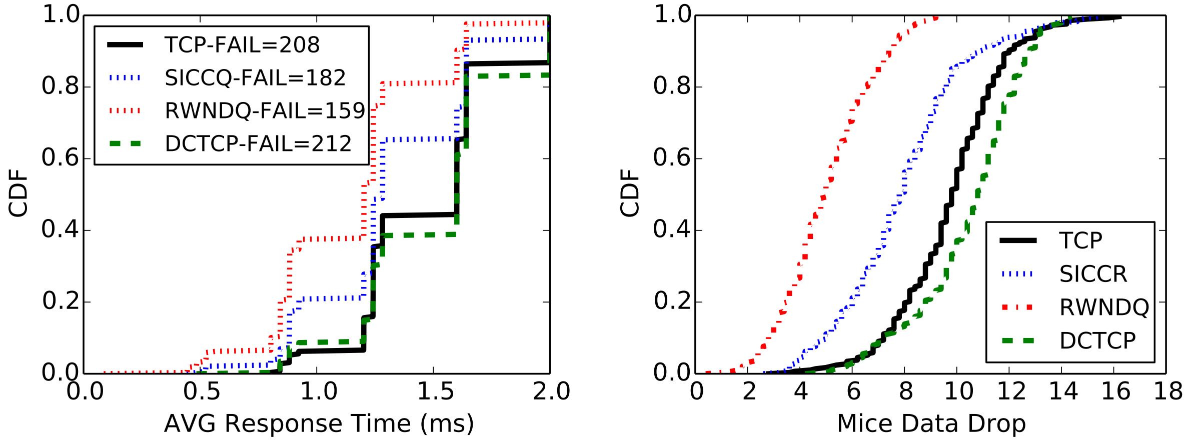
Figure 7: *Performance metrics of TCP, SICCQ, RWNDQ and DCTCP in small fat-tree topology of 144 servers.*

20(TCP)+ 20(IP) + 14(ETH) = 150 bytes message per switch. In total, 1000*150 = 150Kbytes of data would be received by the controller every 5ms, this translates to a bandwidth of 150Kbytes * 8 / 5ms = 240 Mbit/s (i.e., 0.24 Gbit/s). We believe this as a reasonable bandwidth usage for communication overhead between the switches and the controller with respect to the performance gain for the majority of incast flows in datacenters. In addition, in majority of current SDN setups [18], control path is out-of-band (i.e., separate from the datapath network) which helps avoid any added overhead to the datapath used by the servers in the network and the bandwidth is used for data forwarding purposes.

5 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. SICCQ 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 3.2. 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 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.



(a) *Average FCT for mice*

(b) *AVG mice data drop*

Figure 8: *Performance metrics of TCP, SICCQ, RWNDQ and DCTCP in larger fat-tree topology of 288 servers.*

5.1 Testbed Setup

For experimenting with our SICC framework, we set up a testbed as shown in Fig. 10. 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 7 servers 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 are OvS switches which are able to match 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. 10. 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.

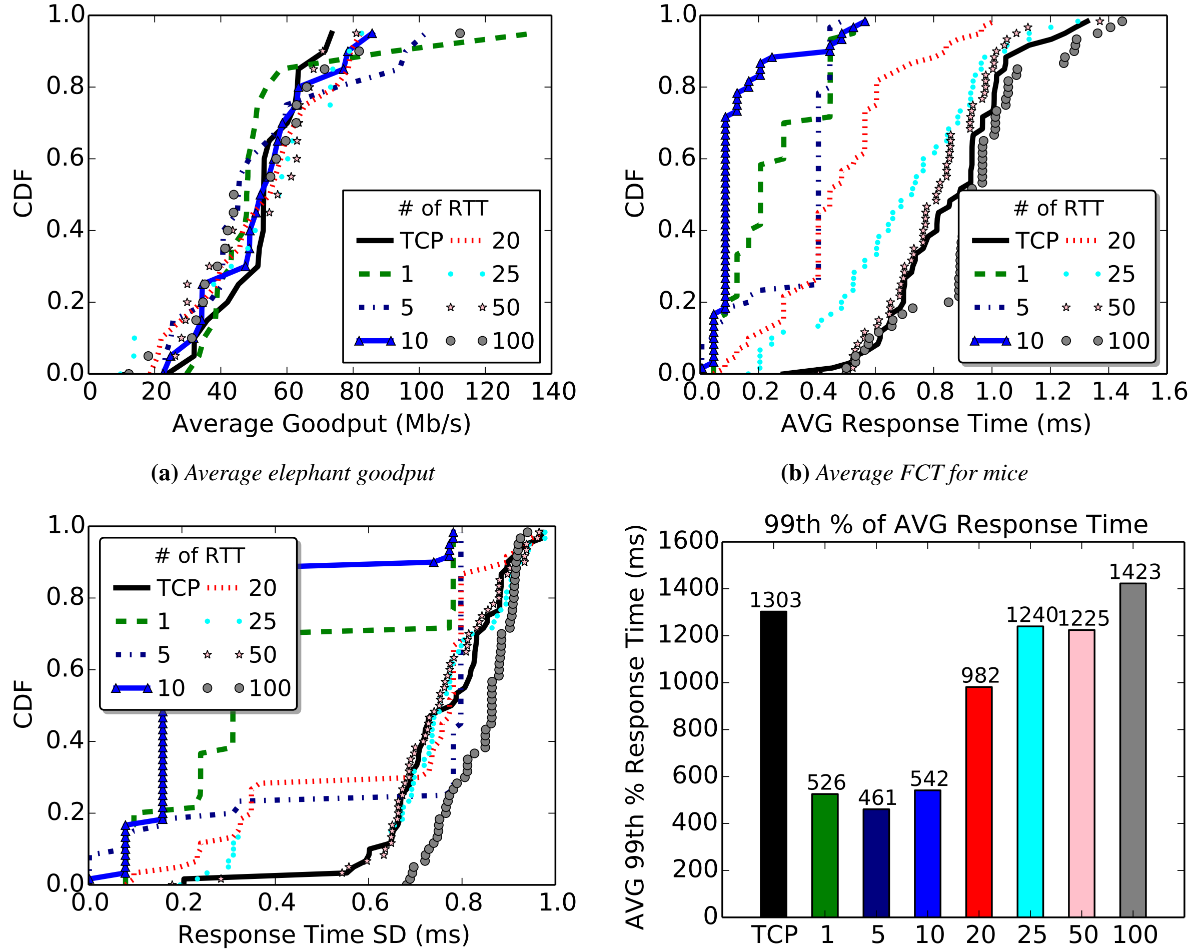
5.2 Experimental Results

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 [6] 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, 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. 11 shows that, in both cases, SICCQ achieves a good balance in meeting the conflicting requirements of elephants and mice. Specifically, Fig. 11a shows that the long-lived elephants are not affected by SICCQ's interruption of their sending rate for a very short period of time after which they restore their previous rates. However, the competing mice flows benefit under SICCQ by achieving a smaller FCT on average with a smaller standard deviation compared to TCP as shown in Fig. 11a and 11c. In addition, as SICCQ 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. 11d.

We repeated the experiment but with 2 iperf flows per sender leading to 42 elephant flows ($7 \times 3 \times 2 = 42$). Fig. 12 shows that SICCQ still achieves a reasonable performance improvement for both TCP NewReno and TCP Cubic. Fig. 12a shows that long-lived elephant flows are not affected by SICCQ. Fig. 12bshows that mice flows still benefit under SICCQ even in a situation where buffers are pressured by the large number of elephants.

[3] The hardware switch running OF-DPA is not following OF1.5 specifications



(a) *Average elephant goodput*

(b) *Average FCT for mice*

(c) *AFCT SD for mice*

(d) *Average 99th % FCT*

Figure 9: *SICCQ with variable queue monitoring interval.*

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

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 mechanisms.

6 Related Work

Over the past few years, research literature became rich with many promising proposals to address the incast congestion problems in datacenter networks. They mainly have been devoted to modifying either or both sides of the TCP stack, proposing new switch mechansim or even relying on SDN's programmability [3, 21, 1, 5, 23, 12, 2] to overcome this critical problem which affect the performance of many cloud applications. In general, these works can be categorized into:

1. **Sender-based**: [21] observed that there was a mismatch between TCP timeout timers in the hosts and the actual round-trip times (RTTs) experienced in DCNs. Typically, when incoming data overflows the small switch buffers, TCP timeouts that last hundreds of milliseconds occur. Due to the design of TCP timeout in most operating systems a latency-sensitive applications that suffers a timeout would have to wait for several hundred RTTs before it can retransmit its data[4]. The proposed solution in [21] modifies the sender TCP stack by using high-resolution timers to enable microsecond-granularity

[4] For example the Linux implementation sets the minimum timeout to 200 millisecond whereas the RTT in a data center ranges typically from a few tens to a few hundred microseconds

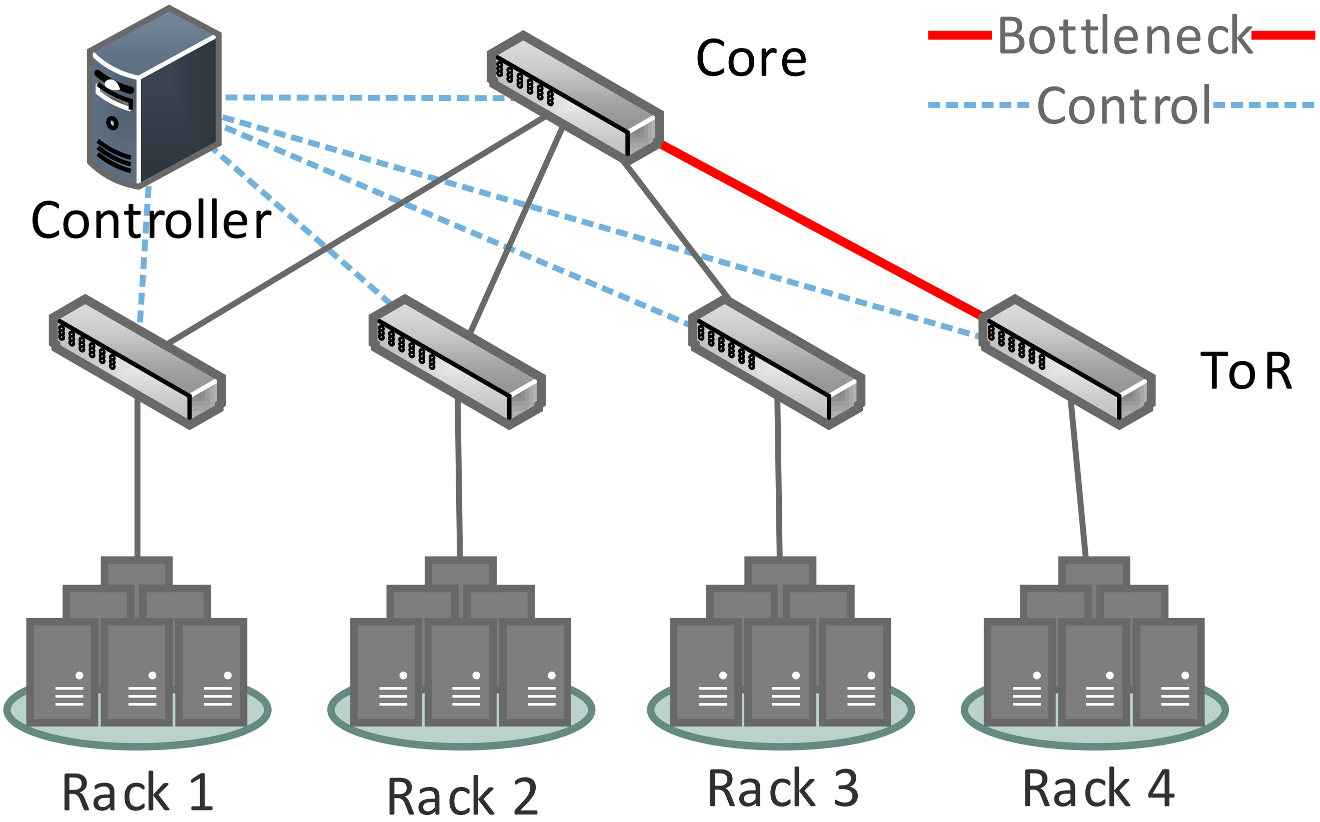
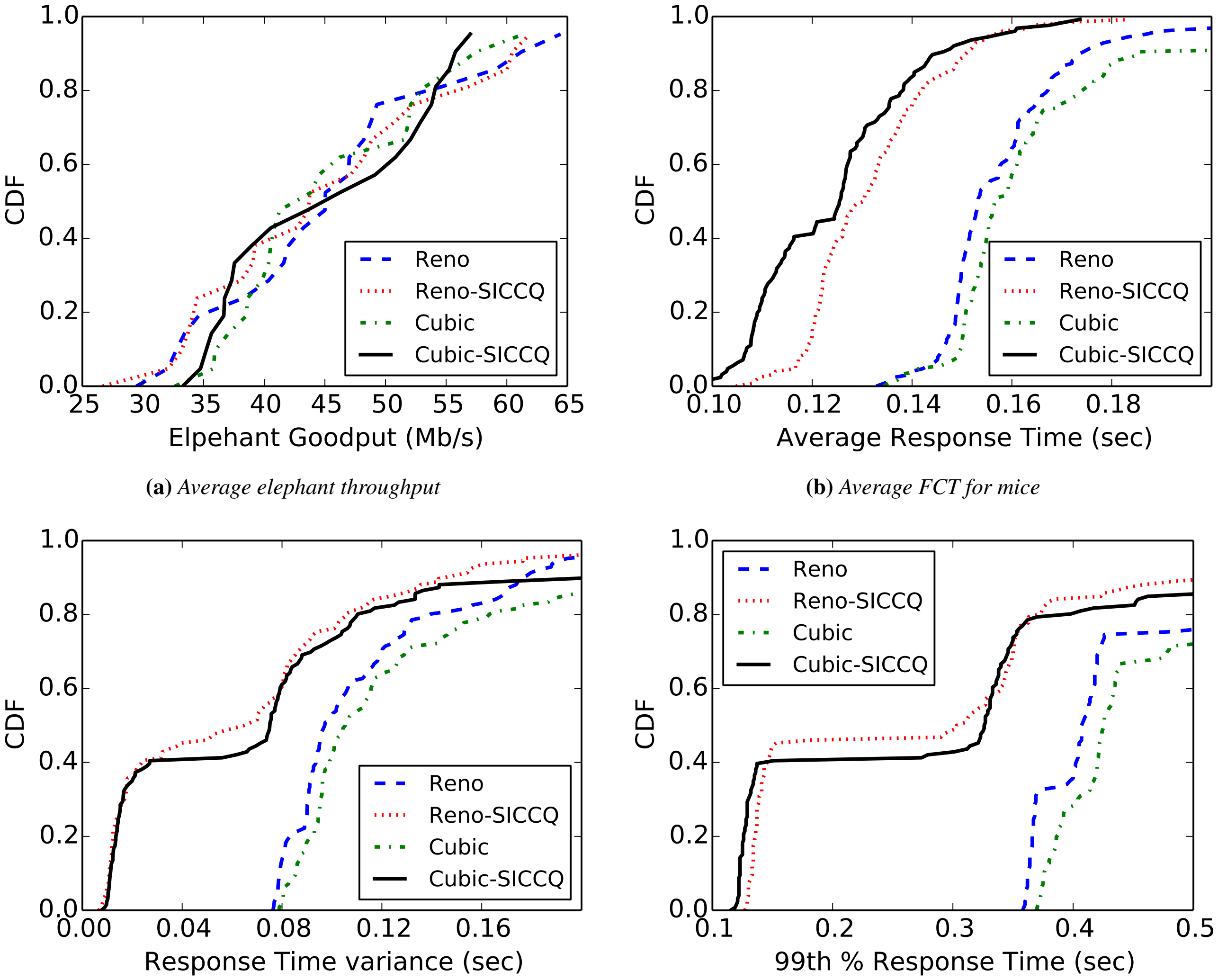


Figure 10: *A real testbed for experimenting with SICC framework*

in TCP timeouts. However, this technique was shown to effectively avoid TCP incast collapse, it may require fine tuning of the Min-RTO based on network conditions and hence a chosen Min-RTO value may not suit all scenarios and network sizes. It also requires the modification and rebuilding of guest VM's TCP.

2. **Receiver-based**: 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 build up issue caused by the co-existence of elephants in the same buffer as the 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**: IQM [2] proposed a small modification to the switches to predict incast events and act proactively. IQM tracks the TCP connection setup and tear-down events at the switch to predict possible incast congestion in the next few RTTs. IQM upon finding that the congestion would be severe and lead to overflow, it sets the receiver window of ACK packets to 1 MSS to slow down the elephants. The simulation results and experiments with IQM in a real testbed show that IQM can almost alienate possible timeouts for TCP incast traffic and achieves a high throughput for TCP elephant traffic. Unfortunately, IQM may not see a big potential for deployment due to the required modification to switch forwarding ASIC. Furthermore, due to IQM's flow-unaware nature, TCP is required to operate without the use of window scaling option. Also, in IQM, TCP flows should be routed through the same forward and backward path which is achievable through SDN functionality.
4. **SDN-based**: SDTCP [12], proposes an incast congestion control mechanism that leverages SDN architecture and implements a network side congestion control. They involve the SDN controller to monitor in-network congestion messages triggered by OpenFlow switches and select currently active elephant flows by inquiring the switches for flow-level information. Then, consequently, the controller will set up OpenFlow rule at the switches. These flow-based rules are set to decrease the sending rate of elephants by rewriting the TCP receive window of ACK packet. The motivation was the global perspective available for SDN controllers, where elephants can be identified and rate-limited during congestion events. The experiments conducted in an emulation environment (Mininet) shows almost zero packet loss for TCP incast



(a) *Average elephant throughput*

(b) *Average FCT for mice*

(c) *SD of FCT for mice*

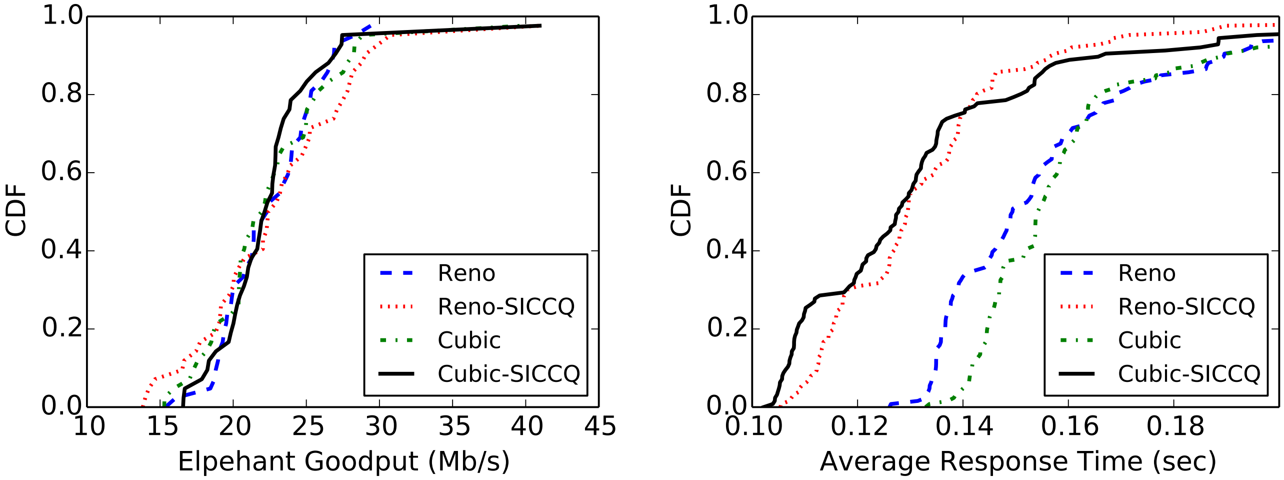
(d) *99th % of FCT for mice*

Figure 11: *SICCQ vs. TCP: 126 mice competing with 21 elephants*

while no great effect on goodput of the elephants. The major problem is that, the proposed modifications i.e., the receiver window rewriting and switches sending out congestion notification messages are unrealistic unless they are implemented and supported by the switching chip.

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 occupancies to infer the start of incast epochs before they start sending 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 the in-place commodity hardware switching. SICC's main contribution is to make a point of principle to not modify the TCP algorithms nor the networking hardware to enable quick and true deployment potential in real operation-critical data



(a) *Average elephant throughput*

(b) *Average FCT for mice*

Figure 12: *SICCQ vs. TCP: 126 mice competing with 42 elephants*

center networks. Further testing of SICC in an operational environment with realistic workloads and scale is necessary.

References

[1] A. M. Abdelmoniem and B. Bensaou. Efficient switch-assisted congestion control for data centers: an implementation and evaluation. In *34th IEEE International Performance Computing and Communications Conference (IPCCC15)*, 2015.

[2] A. M. Abdelmoniem and B. Bensaou. Incast-Aware Switch-Assisted TCP congestion control for data centers. In *IEEE Global Communications Conference (GlobeCom15)*, 2015.

[3] A. M. Abdelmoniem and B. Bensaou. Reconciling mice and elephants in data center networks. In *IEEE 4th International Conference on Cloud Networking (CloudNet15)*, 2015.

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

[5] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data center TCP (DCTCP). *ACM SIGCOMM Computer Communication Review*, 40:63–74, 2010.

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

[7] T. Benson, A. Akella, and D. a. Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM*, 2010.

[8] J. Dean and S. Ghemawat. MapReduce : Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51:1–13, 2008.

[9] W. Eddy. 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] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken. The nature of data center traffic. In *Proceedings of the 9th conference on Internet measurement conference (IMC09)*, 2009.

[12] Y. Lu and S. Zhu. SDN-based TCP Congestion Control in Data Center Networks. In *43th IEEE International Performance Computing and Communications Conference (IPCCC)*, 2015.

[13] N. Mckeown, T. Anderson, L. Peterson, J. Rexford, S. Shenker, and S. Louis. OpenFlow : Enabling Innovation in Campus Networks. *ACM SIGCOMM Computer Communication Review*, 38:69–74, 2008.

[14] NS2. The network simulator ns-2 project. http://www.isi.edu/nsnam/ns.

[15] Open Networking Foundation. SDN Architecture Overview. Technical report, Open Networking Foundation, Dec 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] A. Panda, C. Scott, A. Ghodsi, T. Koponen, and S. Shenker. CAP for networks. In *Proceedings of ACM SIGCOMM HotSDN workshop*, 2013.

[19] A. Rijsinghani. 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] V. Vasudevan, A. Phanishayee, H. Shah, E. Krevat, D. G. Andersen, G. R. Ganger, G. A. Gibson, and B. Mueller. Safe and effective fine-grained TCP retransmissions for datacenter communication. *ACM SIGCOMM Computer Communication Review*, 39:303–314, 2009.

[22] H. Wang, L. Xu, , and G. Gu. Floodguard: A dos attack prevention extension in software-defined networks. In *IEEE/IFIP Conference on Dependable Systems and Networks*, 2015.

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