SICC: SDN-based Incast Congestion Control for Data Centers

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

***Abstract*—Due to the partition/aggregate nature of many distributed cloud-based applications, incast traffic carried by TCP abounds in data center networks. 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-demanding elastic traffic flows in the network worsens the performance degradation further. In this paper, relying on the programmability of Software Defined Networks (SDN), we address this problem in an efficient and easily deployable manner. The proposed SDN-based incast congestion control framework relies on the SDN controller and the hypervisor programmability to solve such congestion problems without altering the guest virtual machines nor the network switches. We assess the performance of the proposed scheme via real deployment in a small-scale testbed and ns2 simulation in larger environments.**

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

I. 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 as **elephants**). In particular, recent studies [1, 2] have shown that while in practice DCNs are crowded with mice, the lion's share in terms of the volume of traffic still goes to the elephants. Furthermore, many mice applications typically generate incast traffic including: *i)* **large-scale data processing applications** where every requested service is broken into parallel tasks assigned to worker nodes such as MapReduce and Spark. The responses from these workers are collected by an aggregation node that finally produces the final result; and *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.

DCNs are structured to provide a high bandwidth and low latency networking environment using Ethernet switches with small buffers 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. Recent works [2, 3] showed that such complex congestion events are inadequately handled by TCP, as it is agnostic to the latency requirements of mice traffic flows and the composite nature of application data. Yet most applications in DCN still rely on TCP for data transport.

Software Defined Networking (SDN) [4] 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 [5] 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.

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. As a result, **the contributions of this work** can be summarized as follows: 1) We develop and build an efficient system namely SDN-based Incast Congestion Control (SICC) that can handle possible contention on the buffer space before the surge of incast traffic; and 2) We demonstrate that SICC can significantly improve the flow completion time compared with TCP, and other alternatives with little impact on the performance of elephant flows; and 3) We evaluate the sensitivity of SICC in face of varying delays between the controller, the switches and the end-hosts.

In the remainder of this paper, we will first discuss our proposed methodology in Section III then present our SDN-based framework and discuss it in Section IV. Then, we will evaluate our framework via ns2 simulations in Section V to compare it to alternative approaches[1]. We finally conclude the paper in VI.

Manuscript is accepted for publication in proceedings of ICC17 ©2017 IEEE. This work is supported in part under Grants: HKPFS PF12-16707, REC14EG03 and FSGRF14EG24

[1]Due to space limitation, implementation details and experimental results of the test-bed deployment can be found in the technical report at [6]. Simulation and implementation source code are available upon request from the authors.

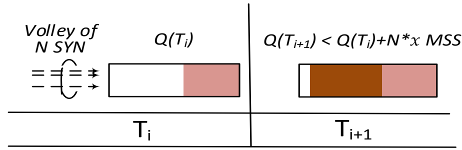


Figure 1: *Before Incast, volley of N SYNs arrive to signal incast at time T_i. It is expected at time T_{i+1} that incast sources increase to the buffer by $N * x * MSS$ bytes*

II. Related Work

Over the past few years, research literature became rich with many promising proposals [2, 3, 7–10] to address the incast congestion problems in datacenter networks. In general, these works can be categorized into:

1) **Sender-based**: in [9], it is observed that there is a mismatch between TCP timeout timers in the hosts and the actual round-trip times (RTTs) experienced in DCNs. Modifying the sender TCP stack to use high-resolution timers was thus proposed to enable TCP timeout detection in the microsecond granularity. DCTCP [3] proposed modifying TCP congestion window adjustment function to react proportionally to the congestion level. RED-AQM parameters are tuned to enforce a small ECN-marking threshold to achieve a small queue length. Both approaches can achieve small delays for mice traffic but require modifications of TCP sender and receiver algorithms as well as fine tuning of RED parameters at the switches for DCTCP.
2) **Receiver-based**: ICTCP [2] 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 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**: The authors in [7, 8, 11] proposed AQM schemes to regulate TCP sending rate with minor modifications to DropTail AQM. RWNDQ [7] tracks the number of established flows to calculate a fair share for each flow. TCP receiver window is updated to explicitly feedback the calculated share to TCP sources. IQM [11] monitors the TCP connection setup and tear-down events at the switch to predict possible incast congestion in the next few RTTs. If congestion is imminent, the receiver window of ACKs are reset to 1 MSS to slow down elephants, making 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 [10] involves the SDN controller to monitor in-network congestion messages triggered by OpenFlow switches and select currently active elephant flows. The controller sets up OpenFlow rule at the switches to decrease the sending rate of elephants via rewriting the TCP receive window of ACKs. The experiments conducted in an emulation environment (Mininet) shows almost zero loss for TCP incast while no great effect on goodput of the elephants. However, the proposed modifications and notification messages from switches are unrealistic unless they are implemented by modifying the switches.

III. Proposed Methodology

A simple illustration of the basic idea of our proposed 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.

SDN framework provides useful statistics on the ongoing number of flows and the queue occupancy for each switch port. Hence, the SDN controller upon forecasting possible incast event, it can send a special message to the sending end-hosts. In turn, end-hosts start rewriting the receiver window ***Rwnd*** in the incoming ACKs to 1 MSS. As a result, since the sending window in TCP is the smallest between the congestion window ***Cwnd*** and ***Rwnd*** , the senders (in particular elephants) are throttled to 1 MSS per RTT[2]. Throttling all flows to a single segment per RTT will enable the congested switches to drain the queue below the congestion threshold. Consequently, ***Rwnd*** rewriting would stop typically after a few RTTs to allow ongoing elephants to recover their previous sending rate (since ***Cwnd*** and ***Rwnd*** are still the same as before). Finally, the controller can roughly estimate N" by simply counting the number of SYNs minus the number of FINs within a tracking interval.

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 module starts to intercept and modify the incoming ACKs for that VM until an incast OFF message is received later or the incast event times out. 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 switches to forward a copy of any SYN/FIN packet through the OpenFlow protocol interface.

IV. SDN-based Incast Congestion Control

The main variables and parameters used in the SICC framework are described in Table I. Notice that T, DM, α_1 and α_2 are algorithm's parameters.

[2]All the rewriting happens at the hypervisor below the VMs to not interfere with the TCP protocol inside the VMs.

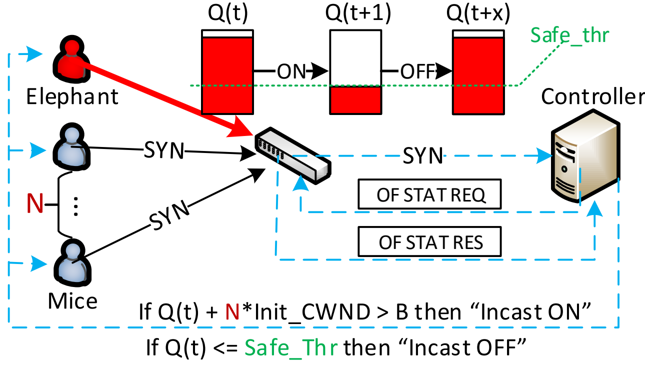


Figure 2: *A detailed view of SICC framework components' interactions which forms a closed-control cycle.*

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

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

A. SICCQ: Incast detection via Queue-based Monitoring

The central controller sets OpenFlow rules at all 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 fixed intervals to calculate a smooth weighted moving average of the queue occupancy. SICCQ shown in Algorithm 1 is event-driven and implements two major event handlers: packet arrivals and incast detection timer expiry to trigger incast on or off message forwarding to the involved sources.

1) **Upon a packet arrival to the switch port:** If this is a SYN packet for establishing a new TCP connection, then the current value of 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 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 worth of bytes TCP_ICWND 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 predicted to overflow in the next interval due to this 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 the senders involved in the incast congestion 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 nominal 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).

Algorithm 1: SICCQ Controller Algorithm

1 **Function** $Packet_Arrival(P, src, dst)$
2   **if** $SYN_bit_set(P)$ **then**
3     $\beta \leftarrow \beta + 1;$
4     $M[src][dst] \leftarrow P.tcpoption.mss;$
5     $W[src][dst] \leftarrow P.tcpoption.wndscale;$
6   **if** $FIN_bit_set(P)$ **then**
7     $\beta \leftarrow MAX(0, \beta - 1);$
8 **Function** $Incast_Detection_Timeout$
9   **forall** sw *in* $SWITCH$ **do**
10   **forall** p *in* $SWITCH_PORT$ **do**
11       $Q[sw][p] \leftarrow \frac{Q[sw][p]}{4} + \frac{3 \times Q[sw][p]}{4};$
12       $\gamma \leftarrow \beta[sw][p] \times TCP_ICWND + Q[sw][p];$
13   **if** $now - \kappa[sw][p] >= DM$ **then**
14   **if** $q[sw][p] \leq (\alpha_1 \times B)$ **then**
15   **forall** dst *in* $PORT_DST$ **do**
16   **forall** src *in* DST_SRC **do**
17               $msg \leftarrow$ "$INCAST\ OFF$";
18   *send msg to src*;
19   **if** $\beta > 0$ *and* $\gamma \geq (\alpha_2 \times B)$ **then**
20   **forall** dst *in* $PORT_DST$ **do**
21   **forall** src *in* DST_SRC **do**
22             $msg \leftarrow$ "$INCAST\ ON$";
23             $msg \leftarrow msg + W[src][dst];$
24             $msg \leftarrow msg + M[src][dst];$
25   *send msg to src*;
26       $\beta[sw][p] \leftarrow 0;$
27   Restart Incast detection timer T;

B. Hypervisor Window Update Algorithm

End-host hypervisors need to be patched and modified to track for any possible messages coming from the SICC 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

Algorithm 2: SICC Hypervisor Algorithm

1 **Function** $Packet_Arrival(P, src, dst)$
2 **if** $INCAST_ON_MSG(P)$ **then**
3 $\kappa[src][dst] = True;$
4 $W[src][dst] = P.wndscale;$
5 $M[src][dst] = P.mss;$
6 **if** $INCAST_OFF_MSG(P)$ **then**
$\kappa[src][dst] = False$;
7 **if** $ACK_bit_set(P)$ **then**
8 $WND_{Scaled} = M[src][dst] >> M[src][dst];$
9 **if** $\kappa[src][dst]$ *and* ***Rwnd*** $(P) > WND_{Scaled}$ **then**
10 ***Rwnd*** $(P) = WND_{Scaled};$
11 Recalculate Internet Checksum for P;

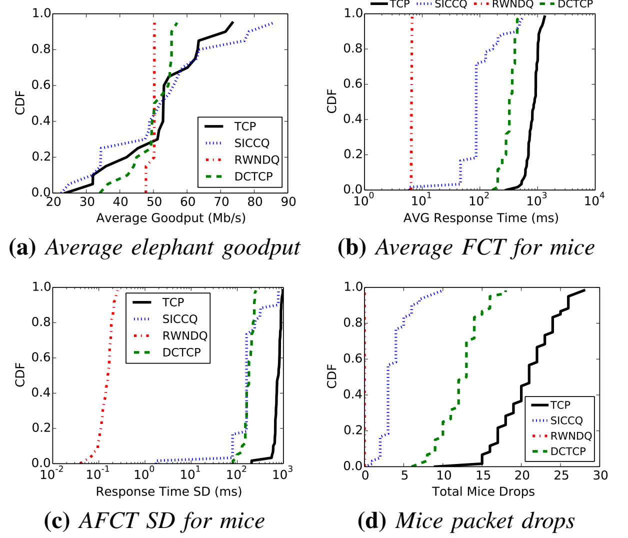
the hypervisor will turn ON the incast flag for this source-destination pair and extracts the attached information about the destination (i.e., the window scale shift factor and the MSS) for ACK receiver window rewriting.

2) **Incast OFF:** If the received packet is identified as an "Incast OFF" message. Then the hypervisor turns OFF the incast flag (i.e., stop ACK update) 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.

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 elephant flows to restart to use their existing congestion window values (that was simply inhibited by the receiver window rewriting) and restore their rate immediately.

C. Practical Aspects of SICC Framework

SICC is a very simple mechanism divided among the SDN controllers and hypervisors of the end-hosts 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 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 efficiently by applying the following straightforward one's-complement add and subtract operations on three 16-bit words: $Checksum_{new} = Checksum_{old} + Rwnd_{new} - Rwnd_{old}$ which 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 [12].



(a) *Average elephant goodput* **(b)** *Average FCT for mice*
(c) *AFCT SD for mice* **(d)** *Mice packet drops*

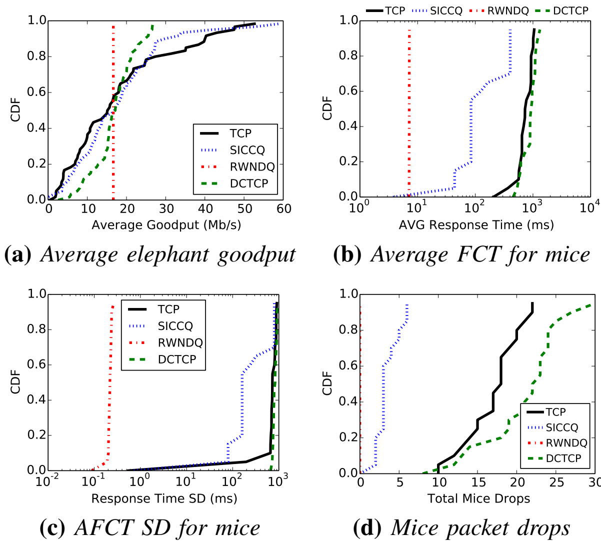
Figure 3: *Performance metrics of TCP, SICCQ, RWNDQ and DCTCP in elephant-to-mice 1:3 ratio scenario.*

SICC framework may be susceptible to performance degradation when subjected to the famous SYN flooding attacks [13]. This attack may lead the senders' window to frequently fluctuate between the current full rate and 1 MSS per RTT. This is a well-known attack which can affect the operation of any TCP flavor, DCTCP and ICTCP as well, because it is targeted towards TCP applications in general. Many proposals have suggested possible solutions to mitigate this attack [13]. SICC can leverage FloodGuard [14] which implements an efficient, lightweight and protocol-independent defense framework for SDN networks. It was shown that it is effective in mitigating flooding attacks while adding only small overhead to the controller.

V. Simulation and Performance Analysis

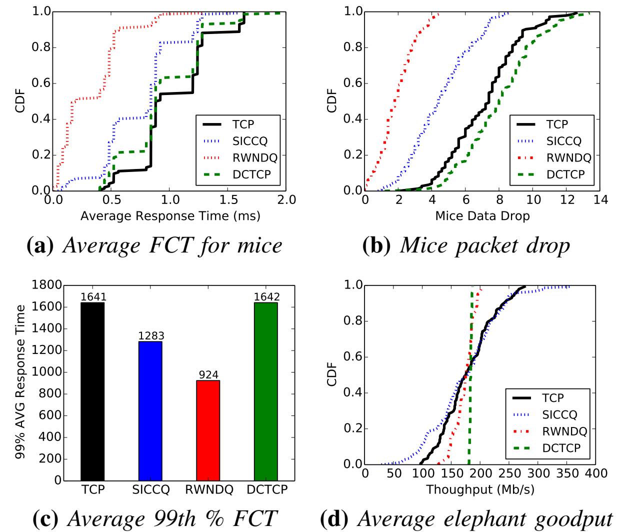
In this section, we study the performance of our algorithm via simulation in network scenarios with a low delay high bandwidth (as is the case in data centers) to compare our system to TCP-DropTail, RWNDQ and DCTCP. The value of α is set to 20% as a safe level of queue occupancy which signals the end of incast epoch and T_i is set to 1ms which is more than the average round trip time in the network. DCTCP parameter K is set to 17% of the buffer size. We use the network simulator ns2 version 2.35 [15], which we have extended with SICCQ framework. In addition, we modified ns2 TCP module to enable TCP flow control of sending receiver window in the ACKs. We use a patch for ns2.35 available from the authors of [16]. 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. The buffer size is 83 packets (125 KB) and max packet size is 1500 bytes.

Single-rooted Topology Simulation: we use a single-rooted topology and run the simulations for a period of 5 sec. We simulate two scenarios to cause incast and queue-buildup situations at the same time. The number of sources is 80 FTP flows. In the first scenario, we simulate an elephant-to-mice ratio of 1:3. We rerun the simulation but increase the



(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 3:1 ratio scenario.*



(a) *Average FCT for mice* (b) *Mice packet drop*
(c) *Average 99th % FCT* (d) *Average elephant goodput*

Figure 5: *Performance metrics of TCP, SICCQ, RWNDQ and DCTCP in small fat-tree topology of 144 servers.*

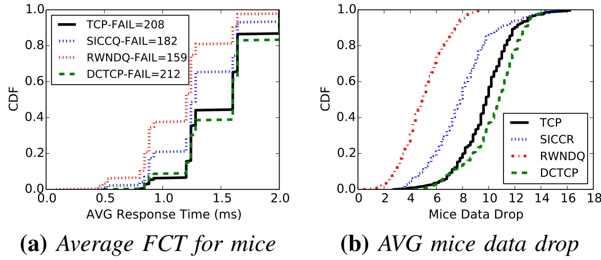
ratio to ratio to 3:1 for elephants to examine how SICCQ would respond in situations where the network is highly loaded with long-lived (background) traffic. All sources start at same time at the beginning of the simulation and elephants keep sending at full link-rate during the whole simulation period. Mice flows who finish their flow quickly close the connection to reopen it at the beginning of each second (5 epochs during the whole simulation). To ensure a relatively tight synchronization between mice flows and create random incast patterns, in each of the five epoch of incast, the individual mice start in a random order with very small inter-arrival time. Each mice flow sends 10KBytes of data then halts until the start of next epoch.

Fig. 3 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. 3a 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. 3b and 3c show that SICCQ can improve mice flow completion time on average with lower variation in response times, achieving a performance close to DCTCP. As expected, RWNDQ is better due to its agility in setting the fair-share of the flows as it is switch-based algorithm and requires switch modification. SICCQ can improve TCP's performance yet requires no modification to the communication end-points nor the switches. Finally, Fig. 3d shows the total cumulative mice 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. 4a suggests that SICCQ again has achieved the same goodput of TCP. Fig. 4b and 4c 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. 4d 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.

Fat-tree Datacenter Topology Simulation: we have created a fat-tree like topology with 1 core, 2 aggregation, and 3 ToR switches with 48 servers/rack to study topologies used in real data center environment. We connected an aggregation server at the last rack which receives the query result from all 144 server in the network. Links from core to aggregation switches have a bandwidth of 10Gb/s, and Links between ToR and aggregation switches have 5Gb/s and finally Server to ToR links have 1Gb/s. This results in a over-subscription ratio of 1:24 at the ToR level. We use propagation delays of 25μs per link and a MinRTO of 200ms. In this scenario, elephants communicate as follows: Rack1→Rack3, Rack2→ Rack3 and Rack3→Rack1. Mice communication defines Racks 1, 2 and 3 to be the workers who send results back to the aggregation server in 5 epochs during the simulation. Fig. 5 shows the results for this scenario. SICCQ is able to improve incast flows FCT compared to TCP and achieves comparable performance as DCTCP with nearly no impact on elephants throughput.

We rerun the simulation but this time in a larger data center setup with 3 aggregation and 6 ToR switches (i.e., 6 Racks) leading to a network of (6×28) 288 servers. Elephant flows are defined as Rack(1,2)→Rack(3,4), Rack(3,4)→Rack(5,6) and Rack(5,6)→Rack(1,2). Fig. 6 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 can be mainly attributed to the reduced mice packet average drop rate and hence the average number of failed flows for SICCQ is reduced as shown in Fig 6a's legend.

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 T defined above in the algorithm. 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: 1, 2, 10, 20, 25, 30, 50, 100 RTTs. Fig. 7 shows the distributions of the mean and variance of FCT for mice, average (99th-percentile) FCT of mice and the average goodput for elephants. Fig. 7a

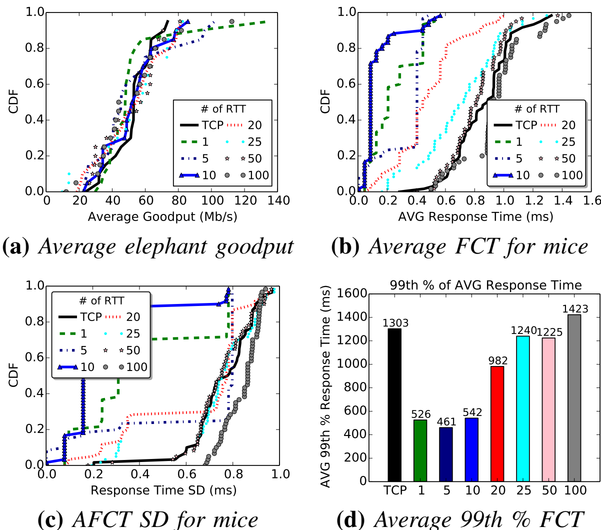


(a) *Average FCT for mice* **(b)** *AVG mice data drop*

Figure 6: *Performance metrics of TCP, SICCQ, RWNDQ and DCTCP in larger fat-tree topology of 288 servers.*

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. 7b, 7c and 7d 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. In typical datacenters, with a minimum RTT of 200-250μs, this translates to reading the queue occupancy once every 4-6.5ms which seems an acceptable probe interval for SDN controllers. This justifies the choice of a monitoring interval of 10 times the RTT in the previous simulations.

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) + 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 most current SDN setups, control path is out-of-band which helps avoid any added overhead to the datapath used by the servers in the network [17].



(a) *Average elephant goodput* **(b)** *Average FCT for mice*

(c) *AFCT SD for mice* **(d)** *Average 99th % FCT*

Figure 7: *SICCQ with variable queue monitoring interval.*

VI. Conclusion and Future Work

In this paper, we proposed a SDN-based congestion control framework to 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 and regularly read the OpenFlow switch queues occupancy to infer the start of incast epochs before flows start sending data into the network. SICC was shown via simulations and testbed experiments to improve the flow completion times for incast traffic without impairing the throughput of elephant flows. A number of detailed simulations showed that SICC can achieve its goals efficiently while outperforming the most prominent alternative approaches. SICC's main contribution is its ability to achieve these improvements without modifying the TCP algorithms nor the networking hardware to enable quick and true deployment potential in real operation-critical data center networks. However, further testing of SICC in an operational environment with realistic workloads and scale is necessary.

References

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

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

[3] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data center TCP (DCTCP)," *ACM SIGCOMM CCR*, vol. 40, pp. 63–74, 2010.

[4] Open Networking Foundation, "SDN Architecture Overview," Open Networking Foundation, Tech. Rep., Dec 2013.

[5] N. Mckeown, T. Anderson, L. Peterson, J. Rexford, S. Shenker, and S. Louis, "OpenFlow : Enabling Innovation in Campus Networks," *ACM SIGCOMM CCR*, vol. 38, pp. 69–74, 2008.

[6] A. M. Abdelmoniem and B. Bensaou, "SDN-based incast congestion control framework for data centers: Implementation and evaluation," CSE Dept, HKUST, Tech. Rep. HKUST-CS16-01, 2016.

[7] ——, "Reconciling mice and elephants in data center networks," in *IEEE Conference on Cloud Networking (CloudNet)*, 2015.

[8] ——, "Efficient switch-assisted congestion control for data centers: an implementation and evaluation," in *IEEE Performance Computing and Communications Conference (IPCCC)*, 2015.

[9] 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 CCR*, vol. 39, 2009.

[10] Y. Lu and S. Zhu, "SDN-based TCP Congestion Control in Data Center Networks," in *Proceedings of IEEE IPCCC*, 2015.

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

[12] opennetworking.org. OpenFlow v1.5 Specification. Https://www.opennetworking.org/sdn-resources/openflow.

[13] W. Eddy. (2007) RFC 4987 - TCP SYN Flooding Attacks and Common Mitigations. Https://tools.ietf.org/html/rfc4987.

[14] 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.

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

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

[17] A. Panda, C. Scott, A. Ghodsi, T. Koponen, and S. Shenker, "CAP for networks," in *Proceedings of HotSDN workshop*, 2013.