Incast-Aware Switch-Assisted TCP Congestion Control for Data Centers

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

***Abstract*—Due to the partition/aggregate nature of many cloud applications, incast traffic is preponderant in data center networks (DCNs). Because TCP is agnostic to this composite nature of the applications traffic and their quality of service requirements, a few congestion events often degrade significantly the user perceived quality of service. This is exacerbated by the co-existence of such incast traffic with other elastic traffic flows in the network. In this paper we address the congestion problems of incast traffic and its interaction with other elastic traffic in DCNs. We propose a switch-assisted TCP congestion control via some small modifications to the switch software that do not require any modification to the TCP protocol nor to the TCP sender or receiver logic. We assess the performance of the proposed scheme via ns2 simulation as well as a real deployment in a small-scale testbed[1].**

***Keywords*—Congestion Control, Data Center Networks, Incast, TCP.**

I. Introduction

Data center networks (DCNs) carry the traffic of a plethora of applications with various traffic characteristics and performance requirements, ranging from a multitude of barrier-synchronized, short-lived, time-sensitive flows (called in the sequel ants) to long-lived, time-insensitive, bandwidth-inclined flows such as backups and virtual machine migration (referred in the remainder as elephants). Recent measurements [1, 2] have shown that in practice DCNs abound with ant type of applications that lead to incast traffic. They can be encountered in *i)* **data-intensive processing systems** such as MapReduce [3] used by web-search, e-commerce, and social networks applications. Such systems handle huge amounts of data by concurrently processing them across many servers. Hence, many-to-many or many-to-one data transfers take place between processing nodes; *ii)* **distributed file systems** where large amount of data are stored in many distributed storage nodes, such as BigTable. When a client retrieves data, parallel access to some of these distributed nodes is needed; and, *iii)* **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. .

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 often 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 [2, 4] to be inadequately handled by TCP, as it is agnostic to the quality of service requirements of ant traffic flows as well as the composite nature of the application data. Yet most applications in DCN still rely on TCP for data transport.

To address such congestion problems in DCNs, recent work has mainly been devoted to modifying TCP to overcome its shortcomings: [5] 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[2]. The proposed solution in [5] consists simply in modifying TCP stack by using high-resolution timers to enable microsecond-granularity in TCP timeouts. This technique was shown to effectively avoid TCP incast collapse. The so-called DCTCP [4] 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 queueing delay). It is shown in [4] that DCTCP can achieve small delays for ants traffic without degrading the link utilization. Nevertheless, DCTCP requires the modification of both TCP sender and TCP receiver algorithms.

ICTCP [2] also proposed 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 ants. 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. Overall,

[1]This work is supported in part under Grants: HKPFS PF12-16707, REC14EG03 and FSGRF13EG14

[2]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

the good results achieved by these prior works are compelling evidence that there is a need for a better way to handle congestion events in DC networks. However, we argue that modifying the TCP protocol and/or the TCP sender or receiver logic is only applicable to small scale private data centers: in most today's public clouds, tenants can upload their own operating system images to their virtual machines. In addition in many instances of applications, the TCP sender and/or receiver are/is outside the cloud and under the total control of the tenant. Our target in this paper is to build 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 dramatically degrade the throughput of elephant flows; (R3) it should not require modification to the TCP sender, nor to the receiver. Any required modification must be in devices that are fully under the control of the DCN operator; (R4) and finally it must be simple enough to be prone to deployment in a real system.

With this objective in mind, we adopt a switch-based approach where the switch actively monitors the occurrence of incast traffic and proactively intervenes, whenever congestion events are forecast, in order to enable ant traffic to pass with minimal congestion. To avoid modifying the existing source and receiver algorithms, our switch uses the TCP flow control window field in the packet headers in a cross-layer approach to temporarily quench the sending rates of elephants without reducing their congestion window sizes, enabling them to recover their sending rates immediately after the incast traffic has avoided congestion.

In the remainder of this paper, we will first discuss our proposed methodology in Section II then present our switch queue management algorithm and discuss it in Section III. We will first evaluate our algorithm via ns2 simulation in Section IV to compare it to alternative approaches, then in Section V we discuss our implementation and evaluation in a small-scale testbed. We finally conclude the paper in VI.

II. Background and Proposed Methodology

A very high level explanation of the rationale of our proposed incast queue management (IQM) algorithm is illustrated in Fig. 1. Considering the persistent queue length in a switch buffer during a measurement period of length T_i to be $Q(T_i)$, if during the period of time a volley of N new TCP connections are established (i.e., N TCP *SYN* packets are seen), it is expected that in the next period T_{i+1}, the queue length $Q(T_{i+1})$ is not more than $Q(T_i) + N * x * MSS$ bytes, where x is the initial window size of TCP in segments. Since incast traffic is ephemeral, the persistent queue is mainly due to elephant flows (or a complex pattern of incast flows arrivals), as a result, IQM measures the number of new flows N by the end of each time interval T_i, and if it predicts the queue length $Q(T_{i+1})$ to reach a congestion threshold in the next interval T_{i+1} then it throttles all the ongoing flows to a sending rate of 1 MSS per RTT each in the next intervals, achieving thus short term fairness between all flows (ants and elephants) and meeting requirement (R1).

In principle, since the TCP source rate is determined by the sending window [6] *Swnd* which is the minimum of receiver

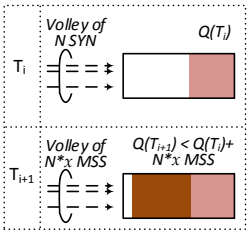


Figure 1: *IQM Rationale: Assuming the queue stable at RTT i, if a volley of N SYN packets is measured in RTT i, it is expected in the worst case that in the next RTT $N * x$ new segments of size MSS will be added to the queue*

window *Rwnd* , and the congestion *Cwnd* , and since *Cwnd* is normally at least equal to 1 MSS, to meet requirement (R3), the switch can rewrite the receiver window field in the TCP ACK headers as a means to throttling the sender rate to 1 MSS per RTT. However this requires the TCP ACKs to travel along the reverse path of the TCP data segments. This can be achieved by invoking software defined networking (SDN) and Openflow as they provide the ability to implement flow-aware forwarding to set the forward and reverse flow along the same path. SDN also provides statistics on the ongoing number of flows and the queue occupancy for each switch port.

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 (if the persistent queue was due to a few elephants), as a result, since incast traffic is ephemeral, to meet requirement (R2), *Rwnd* rewriting would stop after only a few time intervals as soon as the ants finish their transmission, which enables ongoing elephants to recover their previous sending rate (since *Rwnd* is still the same). To meet requirement (R4), instead of tracking individual flow states to estimate accurately the queue length in the next interval, the switch can use rough estimates by simply 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 x to be 1 MSS.

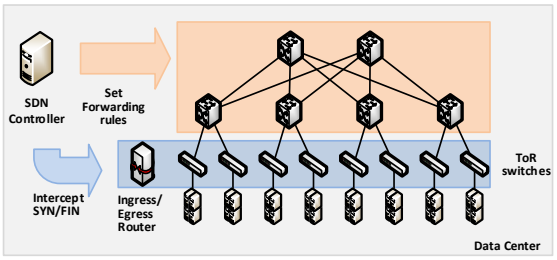


Figure 2: *SDN based implementation scenario of flow-aware network*

Figure 2 illustrates a possible implementation scenario of our proposed mechanism. 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 segments. 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 DC switches must run our IQM algorithm to update the receiver window field in the ACK headers as they cross the switch on the reverse path.

III. Incast-aware queue management algorithm

The main variables and parameters used in the IQM algorithm are described in Table I. Notice that T_i 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.

Table I: *Variables and Parameters used in Algorithm 1*

Parameter name	Description
T_i	Timeout value for Incast monitoring interval
α	No-Incast queue length threshold
Variable name	**Description**
β	Coarsely estimated differential of new connections
Q_{len}	Current length of the output queue
Q_{limit}	buffer size on the forward path
P	a packet
$Rwnd(P)$	Receiver window field in packet P

A. IQM algorithm

Algorithm 1 IQM Algorithm (as an event handler)

switch $(EVENT)$
case $Packet\_Arrival(P)$**:**
  if $Max\_Size < Size(P)$ **then**
    $Max\_Size \leftarrow Size(P)$
  end if
  if SYN_bit_set(P) **then**
    $\beta \leftarrow \beta + 1$
  end if
  if FIN_bit_set(P) **then**
    $\beta \leftarrow MAX(0, \beta - 1)$
  end if
  if ACK_bit_set(P) and Incast_flag **then**
    $Rwnd(P) \leftarrow Max\_Size$
  end if
case $Incast\_Detection\_Timeout$**:**
  if $Q_{len} < \alpha \times Q_{limit}$ **then**
    $Incast\_flag \leftarrow false$
  end if
  $Extra\_traffic \leftarrow \beta \times Initial\_CWND + Q_{len}$
  if $Extra\_traffic > Q_{limit}$ **then**
    $Incast\_flag \leftarrow true$
  end if
  $\beta \leftarrow 0$; Restart Incast detection timer T_i
end switch

IQM algorithm shown in Algorithm 1 is an event-driven mechanism which extends the simple drop-tail queue management with two major event handlers: packet arrivals and incast detection timer expiry to trigger window updates.

1) **Upon a packet arrival:** the maximum segment size seen so far is updated. If this is a SYN packet for establishing a new TCP connection, then the current value of β is incremented, and if this is a FIN packet for closing an established TCP connection, then β is decremented. If the ACK bit is set, the receive window of the Packet is overwritten with 1 MSS worth of bytes whenever the incast flag is set.
2) **Upon elapse of the incast detection timer:** $Extra\_traffic$ indicates the minimal number of extra bytes that will be introduced into the network by the β new and existing connections. Typically each of the sampled new connections starts by sending a full initial congestion window worth of bytes into the network while existing ones will maintain the same persistent queue. If the buffer is expected to overflow in the next interval, then we need to take a fast proactive action to make room for the forthcoming incast traffic, by setting the receive window of passing ACKs in the backward-path to a conservative value of 1 MSS. This will ensure to some extent that the short query traffic (1-10KB) flows will not experience packet drops and hence will not incur the waiting time for retransmission timeout. The incast flag is cleared as soon as the queue length drops below the predetermined threshold enabling thus elephant flows to re-use their previous congestion window values.

Notice that in our algorithm the incast flag tracks imminent congestion rather than incast events per se. But as a by-product it handles incast traffic surges quite well. Indeed, the incast flag can be set because a volley of incast packets is about to arrive and is deemed to lead the buffer to overflow, or the buffer was almost full and a few new arriving (not necessarily incast) flows would lead the buffer to overflow. In both cases our algorithm throttles all ongoing flows to 1 MSS to drain the queue.

B. Practical aspects of IQM Algorithm

IQM maintains a very low loss rate during incast events and enables the switch buffers to absorb sudden traffic surges while maintaining a high utilization. Therefore it is appealing for handling the co-existence of ants and elephants. IQM 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 (ants or elephants) will receive an equal treatment during incast periods. As soon as, the incast traffic, which is short-lived, finishes and leaves the network indicated by the queue occupancy dropping back to less than the predetermined threshold, the elephants immediately restore their previous sending rates by disabling receive window setting to 1 MSS. This typically means during their short existence, incast flows will compete fairly with the elephants and afterwards the elephants can regain their previous steady-state sending rates.

Notice that IQM is a very simple algorithm with very low complexity and can be integrated easily in switches or routers. For example it can be implemented in Linux based routers as a hook in the net-filter framework, to enable modifications to the packet headers prior to their forwarding by IP. This

requires $O(1)$ per packet. IQM 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: $Checksum_{new} = Checksum_{old} + Rwnd_{new} - Rwnd_{old}$ [7]. This also takes $O(1)$ per modified packet. In addition, since IQM is designed to deal with TCP traffic only, monitoring SYN/SYN-ACK and FIN/FIN-ACK bits are simple per-switch-port counters and do not require per-flow information tracking, thus it also requires $O(1)$. As suggested earlier this can also be done by the SDN controller via a set of rules in the ingress/egress and ToR switches or by the switches themselves. Last but not least, in an SDN based DC, the forward and backward routes can be pinned down along the same path by the SDN controller.

All the operations required by our algorithm take $O(1)$ processing time and most importantly involve only the switches and routers under the control of the DC operator. In particular, no modification to the TCP source or receiver is needed.

Our mechanism may raise concerns related to the possibility of being vulnerable to SYN flooding attacks [8]. In our case, the attack would exploit how we set the receive window to 1 MSS whenever a flood of half-open SYN are counted. This behaviour will lead the sender's window to fluctuate each RTT between the current *Cwnd* and 1 MSS. This well-known attack can affect the operation of TCP, DCTCP and ICTCP as well, because it is targeted towards TCP applications in general and many proposals have suggested possible solutions to mitigate this attack [8].

IV. 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 and DCTCP and demonstrate how it outperforms both TCP with Droptail or RED and achieves similar performance as DCTCP without modification to the source and receiver. For IQM, 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 500 μs. 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.

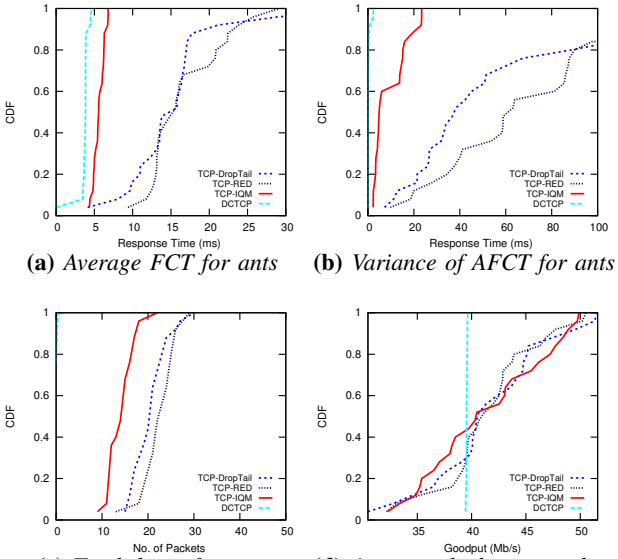
A. Simulation Setup

We used ns2 version 2.35 [9], which we have extended with an IQM module as an inherited class of the DropTail queue management. 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. We compare TCP NewReno with SACK-enabled over Droptail, RED and IQM queue management to DCTCP (which includes a modification of TCP and AQM). For DCTCP, we use a patch for ns2.35 available from the authors [10] 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 500 μs and MinRTO of 2 ms, as opposed to the default 200 ms, which is close to 4 times the average round trip time.

We use a dumbbell topology and run the experiments for a period of 1 sec. The buffer size of the bottleneck link is set in all cases to 83 packets or 125 KBytes where the IP data packet size is 1500 bytes. We simulate two scenarios to cause incast and buffer-bloating situations where the number of sources are 50 and 100 FTP flows respectively. In each scenario, half the sources are elephants and the other half are ants. All sources start at same time at the beginning and while elephants keep sending at full link-rate during the whole simulation period, ants who finish their flow very quickly restart sending for another 5 epochs during the simulation. To ensure a relatively tight synchronization between ant flows, and create an incast traffic scenario, in each of these 5 epochs, the individual ants start in a random order within one packet transmission time. Each ant sends 10KBytes of data and goes to sleep until the next epoch.

B. Simulation Results and Discussion

Fig. 3 and Fig. 4 show the distributions of the mean and variance of the flow completion time (AFCT) for ants, the total number of packet drops from ant flows and the goodput of elephant flows for the two scenarios respectively.

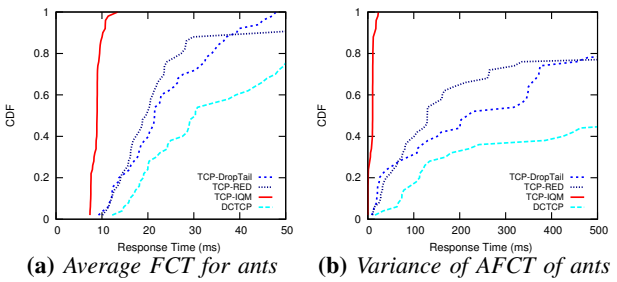


(a) *Average FCT for ants* **(b)** *Variance of AFCT for ants*

(c) *Total drops for ants* **(d)** *Average elephants goodput*

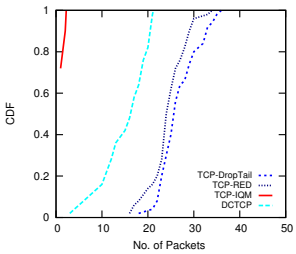
Figure 3: *Performance metrics of TCP (DropTail, RED and IQM) and DCTCP with 50 traffic sources*

According to Fig. 3a, Fig. 3b, Fig. 4a and Fig. 4b, TCP-IQM is able to achieve a shorter AFCT compared to TCP (DropTail, RED) and very close to what DCTCP achieves in the 50 sources case. By increasing the traffic load to 100 sources, TCP-IQM's FCT improves much better than even DCTCP. This is because with 100 TCP sources, the bandwidth-delay product plus the buffer size result in an optimal window of 1 MSS, in this case IQM will not exit from incast state as the queue occupancy is always higher than 20% of the buffer size. Fig. 3c and Fig. 4c show the drop rate from incast traffic with TCP-IQM to be

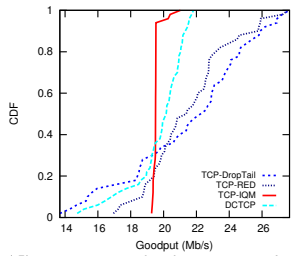


(a) *Average FCT for ants*

(b) *Variance of AFCT of ants*



(c) *Total drops from ants*



(d) *Average elephants goodput*

Figure 4: *Performance metrics of TCP (DropTail, RED and IQM) and DCTCP with 100 traffic sources*

much lower than TCP-(DropTail, RED) and DCTCP in the 100 scenario. TCP-IQM can shield ant traffic from unnecessary drops, which explains the smaller AFCT.

Fig. 3d and 4d show that the average goodput of elephants in TCP-IQM is comparable to TCP-(DropTail, RED) and DCTCP, which indicates that there is no effect on the long-term throughput of these flows.

V. Testbed Implementation of IQM in OpenvSwitch

We further investigate the implementation of IQM as a queue management mechanism based on simple FIFO-DropTail in OpenvSwitch [11] for experimentation in a real-testbed. We patched the Kernel datapath modules of OpenVSwitch with the same functions described earlier in the IQM algorithm. We added IQM functions in the processing pipeline of the packets that pass through the kernel datapath module of OpenVSwitch. In a virtualized environment, IQM-enabled OpenvSwitch can process the traffic for inter-VM, Intra-Host and Inter-Host communications. This is an efficient way of deploying IQM on the host operating system of the switch by only applying a patch and recompiling OpenvSwitch module, making it easily deployable in today's production DCs.

A. Testbed Setup

For experimenting with our patched OpenvSwitch, we set up a testbed as shown in Fig. 5. All machines' internal ports are connected to the patched OpenvSwitch machine shown as switch. We have 5 CentOS machines as destinations and 5 Ubuntu machines as sources all are connected to a different 1 Gb/s D-Link dumb-switch. Similarly, the machines are running the patched OpenVswitch and an Apache web server hosting an **"index.html"** webpage of size 11.5KB. We setup different scenarios to reproduce both incast and buffer-bloating situations with multiple-bottleneck links in the network as shown in Fig. 5.

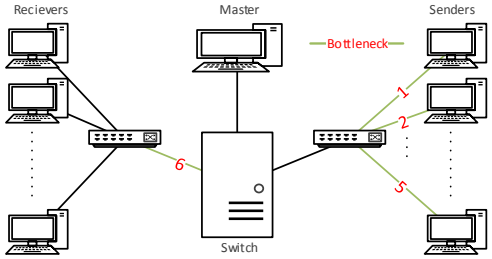


Figure 5: *Testbed scenario for IQM-enabled OpenvSwitch*

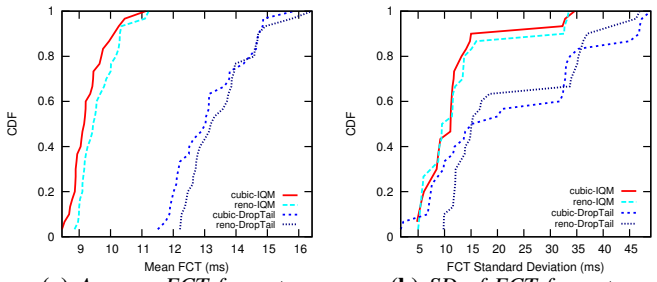
The bottlenecks at the senders are created by creating multiple ports on the OpenvSwitch and binding an iperf flow or an Apache server process to each one of them.

B. Experimental Results

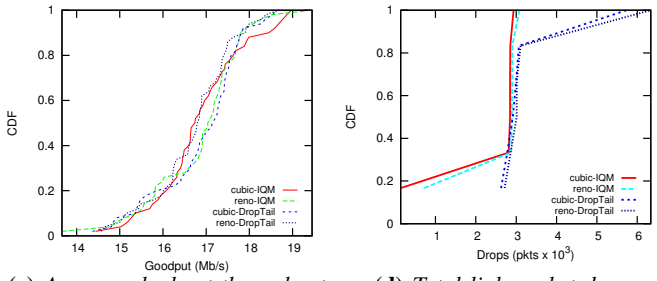
The goals of the experiments are to: *i)* show that with the support of IQM, TCP can support many more connections and maintain high link utilization; *ii)* show that with the support of IQM, TCP can overcome incast congestion situations in the network; *iii)* measure IQM's impact on the FCT of ant flows in incast combined with buffer-bloating situations where ants compete with elephants.

We run an incast with buffer-bloating scenario in which ant traffic competes with elephant flows to see if IQM can help ant flow's AFCT during incast period. We first generate 10 synchronized iperf [12] elephant connections continuously sending for 30 secs from each sender resulting in 50 elephants at link 6. We use Apache benchmark [13] to request **"index.html"** webpage (representing ant flows) from each of the web servers ($6 \times 5 = 30$ in total) running on the same machines where elephants are sending. Note that, we run Apache benchmark, at the 0^{th}sec, requesting the webpage 1000 times then it reports different statistics over the 1000 requests. Fig. 6 shows that, in medium load, IQM achieves a good balance in meeting the conflicting requirements of elephants and ants. The competing ant flows benefit under IQM by achieving a smaller FCT on average with a smaller standard deviation compared to TCP with DropTail as shown in Fig. 6a and Fig. 6b. In addition, as IQM efficiently detects the incast and proactively throttles the elephants, it can decrease the drop rate during incast events, in Fig. 6c, the long-lived elephants are shown to not be affected by IQM's interruption of their sending rate for a very short period. In Fig. 6d, the drops under TCP-IQM is lower than DropTail, which helps ants avoid long timeouts of at least 200ms.

We repeat the experiment, increasing the load to 150 long-lived elephants then introduce the 30 ants. As shown in Fig. 7, IQM is able to satisfy the requirements of latency-sensitive ants eventhough they are outnumbered by elephants. Fig. 7a and Fig. 7b show that ant flows are not blocked by the bandwidth-hogging elephants. The mean FCT and FCT variance under IQM are much smaller than those achieved with DropTail. In addition, Fig. 7c show that, the elephants do not suffer too much from the proactive fairness introduced by IQM during incast periods. From Fig. 7d it is evident that TCP with DropTail is experiencing timeouts due to excessive packet drops which to some extent are avoided under IQM.

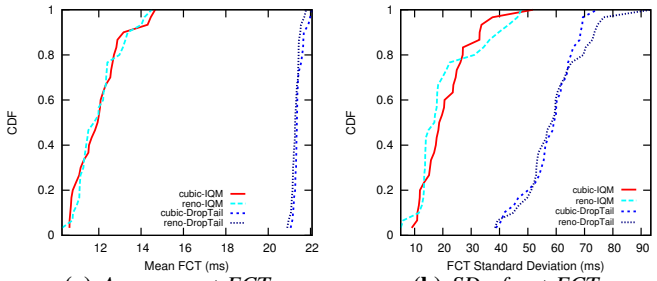


(a) *Average FCT for ants* **(b)** *SD of FCT for ants*

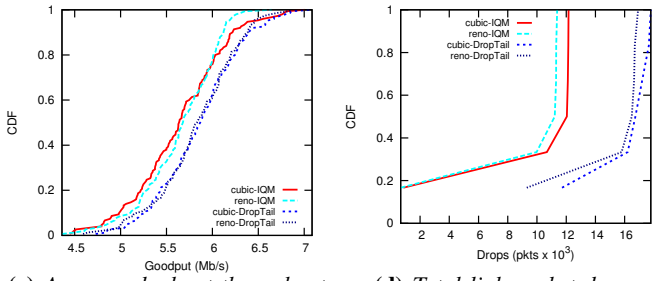


(c) *Average elephant throughput* **(d)** *Total link packet drops*

Figure 6: *Ants FCT for IQM vs. DropTail: 50 elephants competing with 30 ants*



(a) *Average ant FCT* **(b)** *SD of ant FCT*



(c) *Average elephant throughput* **(d)** *Total link packet drops*

Figure 7: *Ants FCT for IQM vs. DropTail: 150 elephants competing with 30 ants*

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

- IQM helps in reducing ant traffic latency and maintains a high throughput for elephants.
- IQM gracefully handles incast events, in low and high load scenarios, while nearly saturating the link.
- IQM achieved all this without any modification to the TCP algorithms at the source nor the receiver and seems to scale well in our testbed.

VI. Conclusion and Future Work

In this paper, we proposed a switch-assisted congestion control mechanism to support the short-lived incast flows, that are known to constitute the majority of flows in data centers. Our mechanism was shown via simulation and testbed experiments to achieve small flow completion times for incast traffic without impairing the throughput of elephant flows. Our IQM mechanism is also shown to be simple, practical and also it meets all its design requirements. A number of detailed simulations showed that IQM can achieve its goals efficiently while outperforming the most prominent alternative approach. Last but not least, knowing that in most public data centers the TCP sender and/or receiver are outside the DC network, IQM makes a point of principle to not modify the TCP algorithms to enable true deployment potential in real networks.

References

[1] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken, "The nature of data center traffic," in *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference - IMC '09*. New York, New York, USA: ACM Press, Nov. 2009, p. 202.

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

[4] 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*, vol. 40, p. 63, 2010.

[5] 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*, vol. 39, p. 303, 2009.

[6] M. Handley, J. Padhye, and S. Floyd. (2000) RFC 2861 - TCP Congestion Window Validation. Https://tools.ietf.org/html/rfc2861.

[7] A. Rijsinghani. (1994) RFC 1624 - Computation of the Internet Checksum via Incremental Update. Https://tools.ietf.org/html/rfc1624.

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

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

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

[11] OpenvSwitch.org. Open Virtual Switch project. Http://openvswitch.org/.

[12] iperf. The TCP/UDP Bandwidth Measurement Tool. Https://iperf.fr/.

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