Enhancing TCP Incast Congestion Control Over Large-scale Datacenter Networks

Lei Xu[1], Ke Xu[1], Yong Jiang[2], Fengyuan Ren[3], Haiyang Wang[4]
l-xu12@mails.tsinghua.edu.cn, xuke@mail.tsinghua.edu.cn, jiangy@sz.tsinghua.edu.cn,
renfy@mail.tsinghua.edu.cn, haiyang@d.umn.edu

[1]Tsinghua National Laboratory for Information Science and Technology, Tsinghua University, Beijing 100084, China

[2]Graduate School at Shenzhen, Tsinghua University, Shenzhen, Guangdong 518055, China

[3]Department of Computer Science & Technology, Tsinghua University, Beijing 100084, China

[4]Department of Computer Science, University of Minnesota Duluth, MN, USA.

***Abstract*—Many-to-one traffic pattern in datacenter networks introduces the problem of Incast congestion for Transmission Control Protocol (TCP) and puts unprecedented pressure to the cloud service providers. To address heavy Incast, we present an Receiver-oriented Datacenter TCP (RDTCP). The proposal is motivated by oscillatory queue size when handling heavy Incast traffic and substantial potential of receiver in congestion control. Finally, RDTCP adopts both open- and closed-loop congestion controls. We provide a systematic discussion on its design issues and implement a prototype to examine its performance. The evaluation results indicate that RDTCP has an average decrease of 47.5% in the mean queue size, 51.2% in the 99th-percentile latency in the increasingly heavy Incast over TCP, and 43.6% and 11.7% over Incast congestion Control for TCP (ICTCP).**

I. Introduction

In datacenter networks, it is known that the many-to-one traffic pattern will also introduce severe *TCP Incast Congestion* problem in the high-bandwidth, low-latency cloud datacenter networks [8], [9]. In *partition/aggregate* architecture, a user request is farmed out among lots of worker nodes, which send back the results almost simultaneously to the aggregators that are responsible for combining data and giving back final results to the user.

When sending data simultaneously to the same receiver, the output queue at the last-hop switch would overflow, causing Incast [10], [11]. In Incast, some flows experience severe packet-drops and Flow Completion Time (FCT) [12]. Worse, Incast creates long-latency flows which miss strict deadline and bring users poor-quality services and enterprises revenue loss [6], [7].

Our aim is to consider numerous concurrent flows under the many-to-one traffic pattern and diverse workloads and satisfy challenges from the upcoming unprecedented large-scale datacenter networks.

In this paper, we take an initial step towards understanding the performance of the existing datacenter-related TCP designs over large-scale datacenter networks. Our experiments indicate that typical TCP protocols fail to work when facing heavier Incast in scaling up data centers. Additionally, we observe receiver's advantages in congestion control.

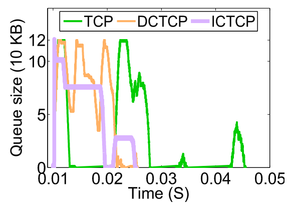
To address higher in-degree Incast, we present Receiver-oriented Datacenter TCP (RDTCP), a protocol that allows the receiver to dominate congestion control. In addition to receiver-dominant control on congestion window, RDTCP leverages an open-loop congestion control, i.e., centralized scheduler, and a closed-loop congestion control, i.e., ECN, together to respond to congestion.[1]

In this paper, we present RDTCP and implement it in ns3 based on TCP-NewReno [1]. Our contributions are as follows: First, we identify the pitfalls of TCP-related Incast congestion control in large-scale datacenter networks and tease out the factors that cause transport deficiency and performance collapse. Second, we propose a hybrid framework which seamlessly integrates the centralized scheduler and ECN at receiver, which is proved efficient to address large-scale Incast.[2]

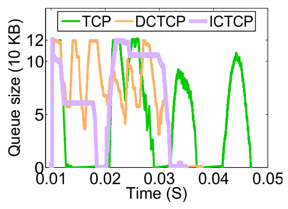
The rest of paper is organized as follows. Section II offers related work; Section III describes the motivations regarding RDTCP; Section IV focuses on the centralized scheduler and ECN of RDTCP. Section V further evaluates the performance of RDTCP in terms of heavy Incast, etc. Finally, Section VI concludes the paper.

[1]Congestion Experienced (CE) bits in IP headers and ECN-Echo (ECE) bits in TCP headers have been used to convey congestion information in ECN packets. We use the term ECN packets as the packets that are marked either with the ECE code point in TCP headers or with the CE code point in IP headers.

[2]Our codes of RDTCP in ns3 are accessible by commanding "svn checkout http://2013rdtcp.googlecode.com/svn/trunk/ 2013rdtcp-read-only".



(a) Queue size with 50 senders.



(b) Queue size with 80 senders.

Fig. 1. Time-series of queue size in the last-hop switch under Incast.

II. Related Work

Congestion control protocols have been developed for many years with the evolution of the Internet. We present typical related protocols in brief. Recently, a chunk of protocols have been provided for datacenter networks. DCTCP is end-to-end protocols based on TCP. It detects congestion degree by ECN packets and makes corresponding congestion behaviors [2]. It effectively suppresses queue buildups in switches and mitigated congestion. Receiver-oriented mechanisms have been proposed for several years [3]. In [3], ICTCP utilizes throughput detection at receiver to improve transport efficiency in data centers. ICTCP is similar to our work but it responds to congestion slowly, as shown in Section V.

III. Motivation

We mimic Incast with 50 and 80 senders which send short flows simultaneously to the same receiver under the same switch at 0.01 s. We set the link delay to 40 μs, link bandwidth to 1 Gbps and switch queue size to 128 KB (i.e., 80 1500-B packets), which conform to real-life data centers. The queue size of three protocols in the last-hop switch are plotted in Fig. 1.

As depicted in Fig. 1(a), in the first round with TCP and DCTCP, 50 packets traverse switch successfully. In the second round, 100 packets with TCP (due to additive-increase) and 90 packets with DCTCP (due to 10 ECN responses and 40 additive-increases) respectively overwhelm the switch queue, leading to packet loss. ICTCP has packet loss in the first round because its initial window is of 2 packets. In subsequent rounds, all the queue sizes suffer from fluctuation. In Fig. 1(b), it is apparent that all the protocols experience fiercer queue oscillation. Even with ICTCP, it has dramatic queue fluctuation at around 0.02 s. The swings of queue size lead to a variable network performance, which, in turn, hampers large-scale deployment of datacenter, not to mention loss in revenue.

The scaling up trend in Fig. 1 enables us to get an in-depth understanding of heavy Incast at the last-hop switch with the increasing number of senders. Because of queue backlog, it is obvious that the queue anomalies and packet-drops become more serious when Incast becomes "large-scale".

The reason for these protocol deficiencies does not lie in themselves, which have been elaborately verified and even applied into real-life datacenter networks. Instead, the key reason lies in the fact that the user demands for web services are constantly growing and the clouds and data centers are continually evolved. With the scaling up data centers, the traditional and typical protocols seem to reach their limits. The problem of how to meet the specific characteristics and expand datacenter transport has become a pressing research issue.

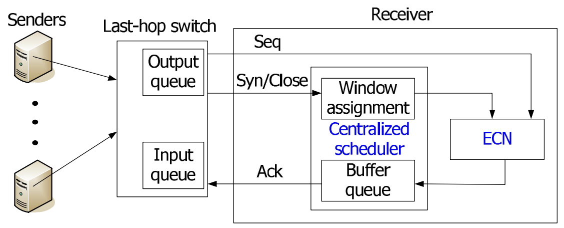


Fig. 2. RDTCP framework.

IV. RDTCP Mechanisms

As portrayed in Fig. 2, RDTCP comprises two main mechanisms: centralized scheduler and ECN. The former aims at eliminating the congestion at last-hop switch while the latter targets on the congestion that occurs elsewhere or are induced by other transport protocols. In this section, we detail the centralized scheduler in Subsections IV-A, IV-B and IV-C. Next we present the relevant components of ECN in the following Subsections IV-D, IV-E, IV-F and IV-G.

A. Preparations for RDTCP

RDTCP is implemented at TCP receiver side. Its receive window size $rwnd$ will be eventually sent off to sender through the advertisement window in TCP header. And the RDTCP sender adopts only $rwnd$ as its unique congestion window to send data. Consequently, the receiver becomes the controller of sender congestion window.

It is advisable that RDTCP utilizes flow priority (e.g., flow size) to allocate congestion window, like [4], [5]. If flow priority is unnecessary, RDTCP still works well by setting all flow priorities to the same value.

In advance, RDTCP has been set a switch queue threshold, $queue\_maxsize$. What RDTCP achieves is to make the runtime queue size in the last-hop switch no greater than $queue\_maxsize$, which uses packet as unit. However, if switch queue threshold is unknown, RDTCP will set $queue\_maxsize$ by $flight\_wins$ when receiver gets first ECN data packet.

As long as the window assignment function has allocated a window size to a flow, the window size given by ECN cannot exceed this value. On the other hand, ECN mechanism counts the flight windows of the flow, which is finally submitted to the buffer queue, as presented in Subsection IV-C. In general, centralized scheduler is globally accessible to all the flows while ECN resides in each connection. In the process of designing RDTCP, the annoying business is to keep consistent congestion behaviors as sender. Fortunately, receiver has consistent sequence number with sender which alleviates the problem.

B. Window Assignment for Each Connection

In RDTCP, when plenty of flows crowd into receiver, i.e., many flows simultaneously transmit into one NIC on the destination host, the window assignment function will partition $queue\_maxsize$ into appropriate window sizes for each flow, by flow priority if necessary.

Algorithm 1 Update Windows for All Flows

```
Input: $info$ //$All\ the\ flow\ information\ has\ been\ stored\ in\ a\ global\ array\ named\ as\ info$;
       $use\_flow\_priority$ //$The\ flag\ to\ use\ flow\ priority$;
 1: $Ascend(info, use\_flow\_priority)$; //$the\ highest\ priority\ is\ at\ the\ tail\ of\ array.$
 2: $i = 0; sum\_win = 0;$
 3: while $i < info.length()$ do
 4:     $info[i].win = info[i].weight * queue\_maxsize;$
 5:     if $info[i].win < 1$ then
 6:         $info[i].win = 1;$
 7:     else if $use\_flow\_priority$ and $info[i].win > info[i].flowsize$ then
 8:         $info[i].win = info[i].flowsize;$
 9:     end if
10:     $sum\_win += info[i++].win;$
11: end while
12: $i = info.length() - 1; N\_noneed\_check = 0;$
13: while $use\_flow\_priority$ and $sum\_win < queue\_maxsize$ do
14:     if $|\ info[i].win - info[i].flowsize\ | <= 1$ then
15:         if $++N\_noneed\_check >= info.length()$ then
16:             $break;$
17:         end if
18:     else
19:         $info[i].win++; sum\_win++;$
20:     end if
21:     if $--i == -1$ then
22:         $i = info.length() - 1; N\_noneed\_check = 0;$
23:     end if
24: end while
```

Implementing the mechanism has three subtleties. 1) The algorithm needs a globally accessible array which stores information of all the active flows from the NIC. To lower storage overhead, the array element only preserves window size and flow priority. It is feasible to deploy the array into device driver layer. 2) Flow priority (e.g., flow size) may be unavailable. Accordingly, the algorithm needs to be robust without flow priority. 3) Due to the fixed flow priority, the function is invoked only when an SYN or CLOSE packet arrives at receiver, as plotted in Fig. 2.

The assignment is in Algorithm 1 which is invoked once SYN packet arrives at receiver. After connection established, the buffer queue mechanism of Subsection IV-C is used to prevent congestion. RDTCP adopts flow priority to partition $queue\_maxsize$[3]. Suppose flow i's priority is p_i and there are n connections in total, then flow i's weight $w_i = p_i / \sum_j^n p_j$. Furthermore, its congestion window is $W_i = w_i \times queue\_maxsize$ (W_i is an integer and takes packet as unit.[4]), as illustrated in Line 4 of Algorithm 1. When W_i is smaller than 1, RDTCP assigns 1 to W_i to avoid stopping flows (Line 6 of Algorithm 1). It is evident that RDTCP is trying to make the sum of all connection congestion windows below $queue\_maxsize$.

In Algorithm 1, two noticeable issues need to be solved. First, the sum of all the connection windows may be smaller than $queue\_maxsize$, resulting in link under-utilization (e.g., with $queue\_maxsize = 40$, 21 flows of the same flow priority ultimately get 21 windows in total, leading to 19 windows idle in the switch queue). This is solved in Lines 13-23 of Algorithm 1. Second, the window size is unnecessarily larger than the flow size (e.g., with $queue\_maxsize = 40$. There are two flows whose sizes are 30 MB and 2 KB, resulting in 1 window and 39 windows, respectively). This is solved in Line 8 of Algorithm 1.

On the whole, the algorithm aims to assign appropriate window sizes for each connection at receiver. However, when the number of connections is too large, it is completely possible for the sum of congestion windows to exceed $queue\_maxsize$ (e.g., when $queue\_maxsize = 40$, each of the 100 flows has been assigned with one window, leading to 100 windows in total). Accordingly, Algorithm 1 does not suffice suppressing congestion. RDTCP needs a complementary mechanism to absorb redundant windows.

Algorithm 1 does not involve too much overhead because it is invoked once each flow. The buffer queue acts as a buffer to absorb redundant packets, so it does not involve computational overhead, as is presented in the following section.

C. The Buffer Queue to Absorb Burtiness

The buffer queue of RDTCP is a queue to store ACK packets of extra connections temporarily. It assures total bytes of flight windows on paths are below $queue\_maxsize$. Since receiver controls congestion window, we add a corresponding counter at receiver to record the flight windows of each flow. When a new data-packet of one flow arrives at receiver, the flow's counter is reduced by the bytes of data-packet. On the contrary, when a new ACK of the flow is sent back,

[3]If flow priority is not available, Algorithm 1 will set all the priorities as a same value.

[4]W_i and $queue\_maxsize$ in Algorithm 1 implicate maximum size of connection, hence their packets mean maximum packets. This dose not impact $flight\_wins$ which uses byte as unit in buffer queue mechanism in Subsection IV-C

the counter will be increased by the increment of ACK-number.[5] If ACK is lost, receiver retransmits ACK and does not counter the bytes, as shown in Subsection IV-F.

The centralized scheduler calculates $flight\_wins$, the sum of counters of all the connections within the same NIC. When $flight\_wins$ is larger than $queue\_maxsize$, the buffer queue will queue up ACK packets instead of sending ACKs until $flight\_wins$ is below $queue\_maxsize$. $flight\_wins$ is accessible by all the connections within the same NIC at receiver. Besides, the buffer queue orders the ACK packets by flow priority. The ACKs with the highest flow priorities are stored near the head of the queue. The priority inversion problem has no chance to happen because the receiver can wait until the queue of last hop switch has enough space to transmit packet and the buffer queue at receiver sends back the highest priority ACK.

Once an ACK is sent back, the queue will be invoked and handle the ACK as input. Since each flow needs a distinct identity to distinguish, the buffer queue stores the thread pointer of flow as flow identity. The procession includes three steps. Firstly, the queue has a unique ACK corresponding to current connection and returns $skiptime$ which denotes the times that an ACK is skipped over (i.e., making no progress in buffer queue).

Second, the mechanism of inserting a current-flow ACK into the queue employs flow priority. However, the queue will degrade into First In First Out (FIFO) queue when flow priority is unavailable. To prevent lower priority flows from being waited in the queue for a long time, the queue uses $skipptime$ for each ACK. When an ACK is skipped over by a higher-priority ACK, its $skipptime$ is added one. When an ACK's $skipptime$ reaches the maximum, it cannot be skipped until it is outputed.

Third, once there is idle space in the switch queue, the buffer queue sends ACKs. The buffer queue plays a vital role in absorbing burtiness from datacenter networks. This mechanism is proved efficient in the face of heavy many-to-one traffic pattern in Section V.

D. Receiver's Basic Congestion Control

The crux of receiver ECN lies in three aspects. First, we imitate TCP NewReno to achieve basic congestion control at receiver. The behaviors of congestion control remain unchanged because we try to make work simple. Second, it is imperative to design an algorithm for receiver to estimate Round Trip Time (RTT), which is of significance to ECN because it has to calculate congestion degree, α_r, and update window every RTT. Finally, retransmission mechanism for ACKs may differ from TCP retransmission. In the following, we present details of surmounting these difficulties.

[5]There are two variables for counting flight windows. One is used for recording increment of ACK number every round, the other is the maximum sequence number expected to arrive. We ignore the details for brevity.

To begin with, the centralized scheduler and ECN cannot contradict with each other. Centralized scheduler targets congestion at last-hop switches while ECN aims at congestion far from receiver. Thus, the former assigns window size W_i while the latter adjusts window size in the range of 1 and W_i.

In addition, receiver's congestion control behaviors resemble TCP NewReno. It includes three parts that are Fast Recovery (Lines 4 to 24), Slow Start (Lines 28 to 30) and Congestion Avoidance (Line 31). seq indicates the sequence number of packet just arrived at receiver, and $expected\ ack$ indicates the sequence number of data that receiver is waiting to receive.

Last, the $duplicated\ ACK\ Count$ is the counter of the seq which is larger than $expected\ ack$. $RxAvail$ denotes the size of received data whose sequence numbers are equal to $expected\ ack$. $RxSize$ indicates the size of all the data received. It is possible that $RxSize$ is larger than $RxAvail$ because the packet of $expected\ ack$ is lost and the following packets are successfully received.

E. RTT Estimator at Receiver

TCP employs Jacobson estimator of RTT and the exponential back-off mechanism at sender while receiver does not. RDTCP ECN has to update congestion window every RTT. For TCP, after sending packet of seq, the sender will estimate RTT when ACK $seq+segment$ arrives at the sender.[6] For RDTCP, we use $rwnd$ of receiver congestion window and current $expected\ ack$ to obtain $expected\ seq$, which is the expected sequence number after RTT. The equation is shown as follow:

$$expected\ seq = expected\ ack + rwnd - segment. \quad (1)$$

Once receiver sends off $expected\ ack$ to sender, it records the sending time as $T1$. Next, when $expected\ seq$ that satisfies Equation 1 arrives at the receiver at time $T2$, the receiver will estimate RTT by $T2 - T1$. The sequence number that is unequal to $expected\ seq$ will not be estimated. RDTCP receiver starts estimating earlier than sender, i.e., in the process of TCP 3-Way Handshake. Further, it estimates RTT only in states of Slow Start and Congestion Avoidance.

F. Delayed ACK and Retransmission

The difference of congestion degree between sender and receiver increases when the delayed ACK time becomes large. In order to make comparison fair, we disable the delayed ACK mechanism at receiver (i.e., sending ACKs every packet).

Importantly, RDTCP leverages receiver's retransmission for ACK and uses RTO^r_{min} to clock ACKs.[7]

[6]We term $segment$ as the maximum size of TCP packet and assume TCP packets are in general of $segment$ size.

[7]Though it is advisable to disable retransmission mechanism at sender, sender retransmission can be conserved. Even with retransmission occurred at sender, the congestion window will not decrease due to receiver's control over congestion behavior.

RTO^r_{min} is set similarly to that of sender. The retransmission timeout is scheduled to fire RTO^r_{min} after not receiving any packets and no corresponding ACK in the buffer queue. Another difference of retransmission between sender and receiver lies in the sending procession. Receiver adds PSH flag in TCP header to force sender to send the packet corresponding to the ACK immediately.

G. ECN Response

In this subsection we present ECN in RDTCP. Like DCTCP [2], RDTCP leverages ECN to detect the extent of remote congestion (e.g., the first-hop switch) and avoid packet loss.[8] The ECN-enabled switch marks all the packets by setting the Congestion Experienced (CE) codepoint [13] when the queue length exceeds a certain threshold K. The RDTCP receiver detects the congestion extent of link by calculating α_r, the ratio of marked packets and all packets over RTT. Afterwards, it decreases window in proportion to α_r as follow:

$$rwnd = rwnd \times (1 - \alpha_r/2). \tag{2}$$

Unlike DCTCP, RDTCP receiver does not need the dual mode state-machine to determine whether to set ECN-Echo bit. Because no matter whether the delayed ACK mechanism is enabled, the receiver is able to directly get α_r and adjust congestion window.

V. Evaluation

We examine RDTCP in terms of queue buildup, goodput, packet-drops and FCT in Incast. We set link delay to 40 μs and bandwidth to 1 Gbps. The size of switch output queue is 128 KB (i.e., 80 1500-B packets), which is akin to commodity switch in real-world data centers. All the protocols are without the delayed ACK mechanism, thus the passive effect of dual state machine of DCTCP will be limited and the comparisons of results are fair.[9] The Retransmission TimeOuts (i.e., RTO^s_{min} for sender and RTO^r_{min} for receiver) are set to 10 ms, which are in line with the parameters in [10]. ECN marking threshold K is set to 40, and $queue\_maxsize$ is set to 40 as well. The switch queue adopts DropTail mechanism. We use flow size as priority. All the parameters remain unchanged in the following subsections unless otherwise noted.

In Incast, many concurrent senders start to transmit 32-KB flows to one receiver at 0.01 s. All the hosts are under the same switch whose configurations are described above. In this section, we need to make sure that RDTCP is immune to effect from this large-scale trend. Hence, in the examination of Incast, we expand the number of concurrent senders from 10 to 100, intending to exhibit the growing inclination. While the number can be further enlarged, we omit the presentation due to space constraints. In any case, the following evaluations does reflect the advantages of RDTCP prominently.

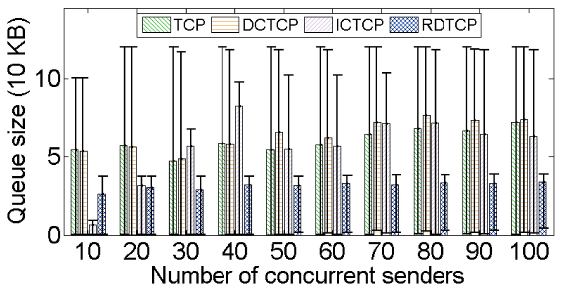


Fig. 3. Queue buildup in the last-hop switch.

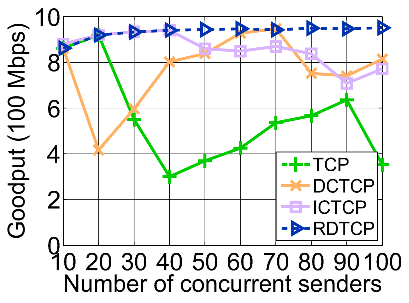


Fig. 4. Goodput and the number of packet-drops and markings of packet in the last-hop switch queue in Incast.

1) Queue Buildup: First and foremost, we focus on the queue size of the switch in the above Incast phenomenon with four protocols. As expected in Fig. 3, RDTCP has the smallest average queue size no matter how many flows are concurrent (except the case of 10 senders where ICTCP takes effect in light-load networks). It has a 47.5%, 50.8% and 43.6% mean decrease in the mean queue size over TCP, DCTCP and ICTCP, respectively.

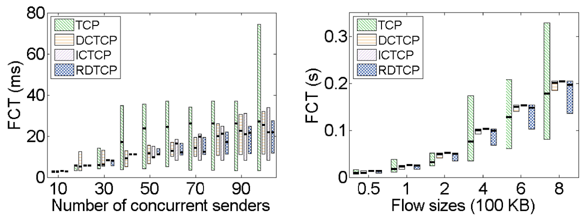
The emphasis is the consistent mean queue size of RDTCP, no matter how many senders there are. In addition, the 95th-percentile queue sizes of RDTCP hold low and stable. All these advantages of RDTCP are the results of centralized scheduler. The buffer queue at RDTCP receiver has restrained total flight windows on the path, keeping the switch queue size at low level. Consequently, RDTCP has the slightest queue fluctuation in all cases.

In contrast, the other protocols are less sensitive to switch queue size, experiencing severe packet loss, as shown in Fig. 3. Because their 95th-percentile queue sizes reach 120 KB which is the maximum switch queue size. All the protocols have similar 5th-percentile queue sizes because at the beginning of transmission the queue sizes are small.

2) Goodput: It is revealed in Fig. 4 that all the protocols except RDTCP suffer from goodput collapse when confronting Incast [11]. With the number of senders increasing, RDTCP achieves almost the best goodput among protocols, which benefits from its zero packet-drops. TCP and DCTCP suffer from significant goodput

[8]We add ECN to TCP/IP model in ns3 of version 3.16.

[9]The sensitiveness to congestion of receiver is not affected when delayed ACK mechanism is disabled.



(a) The 1st, 50th, 99th percentile FCT in Incast.

(b) The 1st, 50th, 99th percentile FCT with varying flow sizes and 30 senders.

Fig. 5. FCTs with varying sender numbers and flow sizes.

decrease, despite afterwards increasing gradually. Note that DCTCP has lower goodput than TCP at 20 senders. This is because its ECN defers congestion to subsequent rounds instead of eliminating them, resulting in more packet-drops.

In Fig. 4, ICTCP behaves consistently with [3] when the number of senders is below 50. However, with further increasing number of sender, ICTCP suffers from goodput collapse. This is because it is oblivious to switch queue size, leading to a sluggish response. As a result, TCP, DCTCP and ICTCP experience serious packet loss, causing timeout at sender. This phenomenon is also termed Block Head TimeOut (BHTO) [12]. However, RDTCP has no such problem because it keeps the the bytes of flight windows on paths below $queue\_maxsize$, i.e., 40.

3) Flow Completion Time: In Fig. 5(a), the 1st, 50th and 99th percentile FCTs of four protocols in Incast are presented. As portrayed in the figure, with the growing number of senders, all the 50th-percentile FCTs increase. It is obvious that RDTCP's 99th FCTs holds the lowest, which is vital in partition/aggregator construction where the last flow determines service quality. RDTCP has 51.2%, 16.4% and 11.7% mean decrease in the 99th-percentile FCT over TCP, DCTCP and ICTCP, respectively. Beyond that, RDTCP has the minimum differences between the 1st and 99th percentile FCTs (except the case of 30 senders), implicating that RDTCP has stable and predictable performance.

Likewise, we construct a set of scenarios where 30 concurrent senders send flows of the same size to one receiver with varying flow sizes while the other configurations remain the same with the Incast scenario. The 1st, 50th and 99th percentile FCTs of four protocols are illustrated in Fig. 5(b).

In Fig. 5(b), RDTCP has the minimum 99th-percentile FCTs. Besides, it achieves the second smallest 1st-percentile FCTs while long flow is sensitive to throughput, i.e., the 1st-percentile FCT. While TCP achieves the minimum 1st-percentile FCTs, its 99th-percentile FCTs are still large due to drastic packet-drops, which is attributed to its proactive congestion control.

VI. Conclusion

The primary contribution of this paper is to present the pitfalls of TCP incast congestion control in large-scale datacenter networks. Viewed in this light, RDTCP, which implements congestion control at receiver, satisfies the requirements of expanding data centers. RDTCP is constructed by two main mechanisms, open- and closed-loop congestion control to cope with different congestion. We orchestrate the mechanisms to respond to congestion reasonably. Further, we pay attention to the serious Incast problem and illustrate that RDTCP is suitable to higher-degree many-to-one communication. In evaluations, RDTCP outperforms other protocols in terms of queue buildup, goodput, FCT, etc.

Acknowledgment

This work has been supported in part by NSFC Project (61170292, 61472212), 973 Project of China (2012CB315803), 863 Project of China (2013AA013302, 2015AA010203), EU MARIE CURIE ACTIONS EVANS (PIRSES-GA-2013-610524) and multidisciplinary fund of Tsinghua National Laboratory for Information Science and Technology.

References

[1] Ns3. http://www.nsnam.org/.

[2] M. Alizadeh, A. Greenberg, D. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "DCTCP: Efficient Packet Transport for the Commoditized Data Center," *Proc. ACM SIGCOMM'10*, pp. 63-74, Aug. 2010.

[3] H. Wu, Z. Feng, C. Guo, and Y. Zhang, "ICTCP: Incast Congestion Control for TCP in Data Center Networks," *IEEE/ACM Trans. Networking (TON)*, vol. 21, no. 2, pp. 345-358, Apr. 2013.

[4] B. Vamanan, J. Hasan, and T. N. Vijaykumar, "Deadline-Aware Datacenter TCP (D^{2}TCP)," *Proc. ACM SIGCOMM'12*, vol. 42, no. 4, pp. 115-126, Oct. 2012.

[5] A. Munir, I. A. Qazi, Z. A. Uzmi, A. Mushtaq, S. N. Ismail, M. S. Iqbal, and B. Khan, "Minimizing Flow Completion Times in Data Centers," *Proc. IEEE INFOCOM'13*, pp. 2157-2165, Apr. 2013.

[6] N. Dukkipati and N. McKeown, "Why Flow Completion Time is the Right Metric for Congestion Control," *ACM SIGCOMM Computer Communication Review*, vol. 36, no. 1, pp. 59-62, Jan, 2006.

[7] J. Brutlag, "Speed Matters for Google Web Search," http://code.google.com/speed/files/delayexp.pdf, 2009.

[8] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107-113, Jan 2008.

[9] M. Isard, M. Budiu, Y. Yu, A. Birrell and D. Fetterly, "Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks," *Proc. EuroSys'07*, vol. 41, no. 3, pp. 59-72, June 2007.

[10] V. Vasudevan, A. Phanishayee, H. Shah, E. Krevat, D. Andersen, G. Ganger, G. Gibson, and B. Mueller, "Safe and Effective Fine-grained TCP Retransmissions for Datacenter Communication," *Proc. ACM SIGCOMM'09*, vol. 39, no. 4, pp. 303-314, Oct. 2009.

[11] R. Griffith, Y. Chen, J. Liu, A. Joseph, and R. Katz, "Understanding TCP Incast Throughput Collapse in Datacenter Networks," *Proc. ACM WREN*, pp. 73-82, Aug. 2009.

[12] J. Zhang, F. Ren and C. Lin, "Modeling and Understanding TCP Incast in Data Center Networks," *Proc. IEEE INFOCOM'11*, pp. 1377-1385, Apr, 2011.

[13] K. K. Ramakrishnan and S. Floyd, "The Addition of Explicit Congestion Notification (ECN) to IP," *RFC 3168*, Sep. 2001.