Computer Networks 124 (2017) 46–60

Contents lists available at ScienceDirect

Computer Networks

journal homepage: www.elsevier.com/locate/comnet

Throughput optimization of TCP incast congestion control in large-scale datacenter networks

Lei Xu[a], Ke Xu[a,*], Yong Jiang[b], Fengyuan Ren[a], Haiyang Wang[c]

[a] Department of Computer Science & Technology, Tsinghua University, Beijing, China
[b] Graduate School at Shenzhen, Tsinghua University, Shenzhen, Guangdong, China
[c] Department of Computer Science at the University of Minnesota Duluth, MN, USA

ARTICLE INFO

Article history:
Received 25 May 2016
Revised 29 May 2017
Accepted 5 June 2017
Available online 6 June 2017

Keywords:
Datacenter networks
Transport protocol
Switch queue
Incast
Congestion control

ABSTRACT

The many-to-one traffic pattern in datacenter networks leads to Transmission Control Protocol (TCP) incast congestion and puts unprecedented pressure to cloud service providers. The abnormal TCP behaviors in incast increase system response time and unavoidably reduce the applicability of cloud-based system deployments. This paper proposes Receiver-oriented Congestion Control (RCC) to address heavy incast in large-scale datacenter networks. RCC is motivated by oscillatory queue size of switch when handling heavy incast and substantial potential of receiver in congestion control when using TCP. RCC makes effective use of centralized scheduler and Explicit Congestion Notification (ECN) at receiver. The RCC prototype is realized in network simulator 3 (ns3) which implements TCP exactly. This paper details the RCC design and evaluates its performance in diverse and heavy workloads. The evaluation results indicate that RCC has an average decreases of 47.5% in the mean queue size and 51.2% in the 99th-percentile latency in the heavy incast over TCP.

1. Introduction

The emergence of cloud computing as an efficient means providing computation can already be felt with the burgeoning of cloud-based applications. Such systems as iCloud, Dropbox, Facebook and Amazon EMR have enjoyed phenomenal growth over the past few years. For instance, Facebook announced that they had Hadoop clusters with 100 petabyte (PB) data across more than 50,000 servers [1]. Other systems as Azure, Dropbox, iCloud are also attracting an increasing number of users and scaling their system deployments on datacenter networks [2,3]. The trend of large-scale datacenter networks is irresistible.

The datacenter networks in this paper are regarding wired networks instead of wireless datacenter networks [4]. The ubiquitous many-to-one traffic pattern in datacenter networks poses challenges for Transmission Control Protocol (TCP). As illustrated in Fig. 1, the many-to-one traffic pattern occurs on the *partition/aggregate* architecture where many work nodes transmit data to the aggregator node. This can easily cause the *TCP incast Congestion* problem in datacenter networks [5–8]. In incast, data flows experience severe packet-drops and long Flow Completion Times (FCTs), bringing users poor Quality of Service (QoS) and enterprise revenue loss [9–11]. It is necessary to design protocols according to network environments [12].

To mitigate this problem, network researchers have proposed effective protocol designs, such as Incast congestion Control for TCP (ICTCP) and Data Center TCP (DCTCP) [13,14]. DCTCP is a pioneer of datacenter transport protocols. Using the Explicit Congestion Notification (ECN) mechanism, DCTCP suppresses queue buildups and packet-drops [13]. H. Wu et al. proposed ICTCP, which controls congestion windows by monitoring receiver throughput. It is the first attempt to control rates of flows at receiver [14]. These protocols are effective for common incast instead of heavy incast which is explained in Section 3.1.

This paper designs a novel transport protocol in congestion control for heavy incast to satisfy requirements from large-scale datacenter networks. Firstly, we investigate heavy incast and its cause. Second, we propose Receiver-oriented Congestion Control (RCC) based on TCP, a mechanism that allows receiver to dominate congestion control. RCC leverages both an open-loop congestion control, i.e., centralized scheduler, and a closed-loop congestion control, i.e., ECN, at receiver to respond to congestion. On the one hand, in RCC, ECN is deployed at receiver to achieve normal

* Corresponding author.
E-mail addresses: l-xu12@mails.tsinghua.edu.cn (L. Xu), xuke@mail.tsinghua.edu.cn (K. Xu), jiangy@sz.tsinghua.edu.cn (Y. Jiang), renfy@mail.tsinghua.edu.cn (F. Ren), haiyang@d.umn.edu (H. Wang).

http://dx.doi.org/10.1016/j.comnet.2017.06.004
1389-1286/

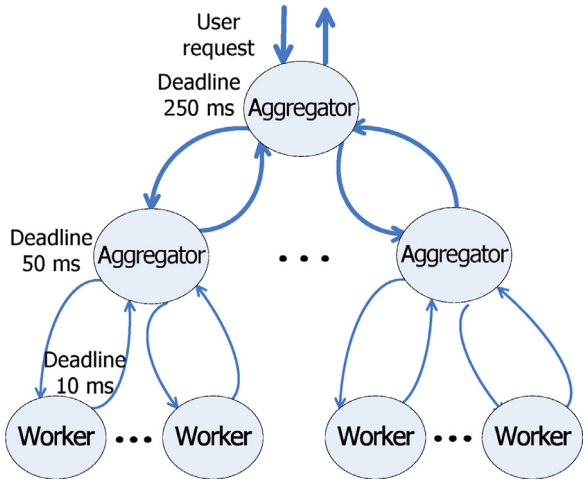


Fig. 1. Partition/aggregate architecture.

congestion control in datacenters [13].[1] On the other hand, the centralized scheduler is used to suppress the burstiness of incast. The previous work of RCC is in [15] and this paper details its design and evaluations.

Integrating the open- and closed-loop congestion controls at receiver is challenging for two reasons. First, the congestion control at receiver has to be compatible with that of TCP. Further, they belong to different types in terms of congestion mechanisms. This paper shows that by arranging the above two congestion controls in a reasonable order, the receiver coordinates different congestion decisions effectively.

Our contributions are as follows:

- This paper teases out the factors impacting transport performance and identifies the root reasons of incast congestion in large-scale datacenter networks.
- We design the receiver congestion control and combine open- and closed-loop congestion controls at receiver. To the best of our knowledge, this is the first attempt to improve transport protocols by combining two congestion controls at receiver.
- We provide a prototype of RCC and evaluate its performance in network simulator 3 (ns3).[2]

The rest of this paper is organized as follows. Section 2 offers related work; Section 3 describes the motivations of RCC; Section 4 details the design of RCC. Section 5 details the analysis of RCC factors. Section 6 further evaluates the performance of RCC in terms of heavy incast, etc. Finally, Section 7 concludes the paper.

2. Related work

Congestion control protocols have been developed for many years with the evolution of the Internet. Network researchers have been improving transport protocols to suit different circumstances all the time. The majority of protocols spring up based on Transmission Control Protocol (TCP) [16]. The state-of-the-art TCP makes effective use of bandwidth by adjusting window sizes according to the Additive-Increase and Multiplicative-Decrease (AIMD) approach. The following presents typical TCP-based protocols in brief.

TCP Reno is proposed to improve TCP throughput when encountering a packet loss [17]. Further, TCP NewReno adds an algorithm for partial Acknowledgment (ACK) during the fast recovery phase for the same aim [18]. TCP Vegas achieves better throughput than TCP Reno. It employs several novel techniques, e.g., Spike Suppression and Congestion Detection by throughput [19]. To meet the challenges from large Bandwidth and Delay Product (BDP) networks, CUBIC is introduced with a window growth function, which is a cubic function of the elapsed time from the last congestion event [20]. Another direction of TCP development is code-based TCP which is beyond the scope of this paper [21,22]. The above protocols mainly focuse on Internet backbone networks rather than specified ones such as datacenter networks.

A chunk of protocols have been proposed for datacenter networks. DCTCP detects congestion degree by ECN packets and makes corresponding congestion decisions [13]. DCTCP has suppressed queue buildups of switches and mitigated congestion. D^3 has introduced deadline into congestion solutions, and it works well with deadline flows [23]. Further, D^2TCP adds flow deadline to DCTCP congestion window control [24]. L^2DCT advances D^2TCP by deploying deadline into not only window decreasing but also increment [25]. PDQ is proposed completely for flows with priorities, achieving an excellent datacenter transmission [26]. Meanwhile, pFabric employs priority switch queues and simplified TCP to achieve outstanding datacenter transport [27]. PASE synthesizes existing transport strategies to suit the need of datacenter networks [28]. Recently, TIMELY is designed based on the Round Trip Time (RTT) mechanism [29].

Receiver-oriented mechanisms have been proposed for several years. There are typical implementations for different contexts [14,30,31]. ICTCP utilizes throughput detection at receiver to improve transport efficiency of datacenter networks in [14]. TCP-Real leverages receiver's ability of decoupling packet loss from window adjustments to avoid unnecessary back-offs [30]. TCP-RTM makes receivers ignore unimportant packet loss for multimedia applications in [31]. Recent work [32] also leverages receiver's ability on detecting actual throughput on the attached link. These procotols reflect the feasibility to deploy congestion control at receiver and receiver's potential of congestion control.

A notable receiver-oriented proposal to cope with incast congestion is PAC [33]. Through a proactive ACK control, PAC is able to cope with heavy incast in datacenter networks. PAC has proved that receiver-oriented method is effective in solving incast problems. The main difference between PAC and RCC is that PAC does not touch TCP-related protocols while RCC is completely a TCP-related protocol.

The following section describes the motivations and objectives of RCC.

3. Motivations

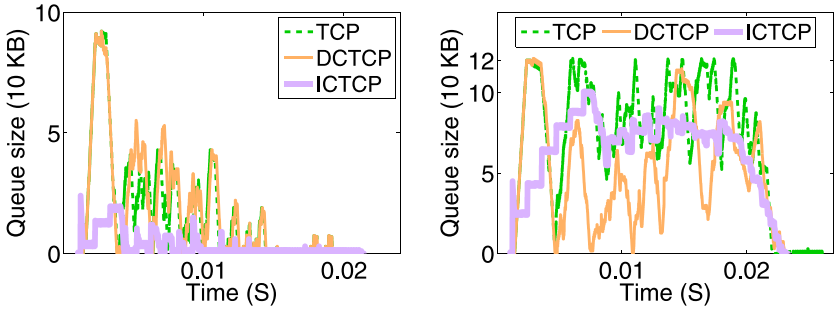
3.1. Queue buildup and packet loss

To analyze heavy incast, this section mimics incast with 50 and 80 senders sending short flows to the same receiver. All the hosts are connected to an 81-port switch whose queue size is 128 KB (i.e., 80 1500-B packets). The starting time of 80 senders complies with exponential distribution with the mean of 15 ms. Hence they are called as randomized senders. Each sender sends 32 KB flow. The link delay is 40 μs and the link bandwidth is 1 Gbps. The real time size of the output queue in the last-hop switch are plotted for three protocols, TCP, DCTCP and ICTCP, in Fig. 2(a) and (b).

In the randomized scenarios in Fig. 2(a) and (b), the queue oscillations are severe. With the increasing number of senders, i.e., 80 senders, oscillation occurs more frequently and fiercely in Fig. 2(b). These two figures show that with increasing senders,

[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 to describe the packets that are marked either with the ECE code point in TCP headers or with the CE code point in IP headers.

[2] The C++ codes of RCC in ns3 are fully accessible at https://github.com/thuxl/rdtcp.



(a) Queue size with 50 randomized senders. (b) Queue size with 80 randomized senders.

Fig. 2. Motivation of queue size.

queue backlogs make queue anomalies and packet-drops more serious.

In datacenter networks, incast is caused by many-to-one traffic patterns and shallow buffered switches. For one thing, barrier-synchronized communications such as MapReduce [5] and Dryad [6] are easy to lead to a many-to-one traffic pattern. For another, in large-scale datacenter networks, the majority of switches are shallow buffered. In this paper, heavy incast has harsher condition than common incast. In heavy incast, all the senders send data *simultaneously* in a many-to-one traffic pattern with shallow buffered switches. Sending data simultaneously is a hasher condition than randomized senders in Fig. 2(a) and (b). Even the number of senders is around 100, the incast degree is heavy or reveals "large-scale" trend.

3.2. The advantages of receiver

In this section, we chose DCTCP to illustrate receiver's advantages on congestion control. We use α from DCTCP to estimate the fraction of packets that have ECN marks. α is updated once for every window of data, i.e., roughly one Round Trip Time (RTT), as follows:

$$\alpha \leftarrow (1-g) \times \alpha + g \times F, \tag{1}$$

where F is the fraction of packets that were marked in the last window of data, and $0 < g < 1$ is the weight given to new samples against the past in the estimation of α.

In the experiment, four long-lived flows are sent by four hosts to the same receiver by DCTCP. All the four hosts connect to four 1-Gbps-ports of the same switch where each link has a 40-μs delay. The switch queue has capability of 80 1500-B packets. The switch queue has the ECN funtion with a 40-packet ECN threshold. A dual-state machine is added to determine whether to set ECN-Echo bit according to [13].[3] To make comparison fair between sender and receiver, DCTCP updates congestion window every 200 μs instead of one RTT. Also, we add an ECN packet counter at receiver which calculates congestion degree as well. All the flows start at the same time.

The α of the first flow is plotted in Fig. 3(a) and (b) to demonstrate the differences of α between sender and receiver with delayed ACK set to 1 and 3, respectively. The situation of the other three flows are similar. When the real-time queue size is larger than 40 packets, it will trigger receiver and sender to calculate α, separately. However, the value of α between receiver and sender is non-consistent in the same link situation, especially when the delayed ACK is set to 3 in Fig. 3(b).

The reason for a higher receiver α in Fig. 3(a) is that the dual-state machine at receiver will not immediately send back ECN ACK packets after receiving CE-marked data packets. Instead, the receiver still sends a non-ECN ACK back to the sender, making the sender with one ECN packet less calculated than the receiver. Further, when congestion is relieved, i.e., normal data packets start to arrive at receiver, one ECN ACK as a delayed ECN packet will be sent to sender, which generates a smaller value of α at sender despite no congestion on the link. This discrepancy will be enlarged when the number of delayed ACK increases, as depicted in Fig. 3(b) before 0.2 s. After that, the value of sender's α is higher than that of receiver's because after congestion is relieved, at most 3 ECN ACKs are still received by sender, leading to an overestimated α.

Beyond that, the receiver can aggregate the information received over all incoming TCP flows, and control appropriately. In conclusion, receiver has more accurate congestion information than sender and has natural advantage of aggregating the information for all incoming TCP flows, making it suitable to many-to-one traffic patterns in large-scale datacenter networks.

4. The design of RCC

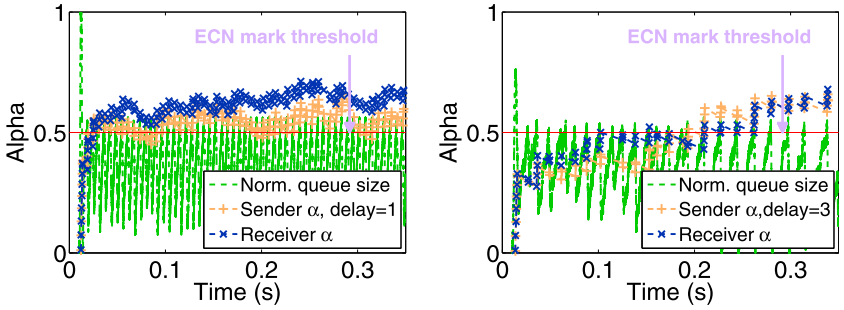
As shown in Fig. 4, RCC is implemented at the receiver side. The centralized scheduler consists of window assignment that allocates the maximum window size for a flow and buffer queue that absorbs burstiness and counts the total number of flight windows. In addition, ECN works as that in sender. The receive-window size *rwnd* will be eventually sent off to sender by the advertisement window in ACK packet. And the sender adopts *rwnd* as its unique congestion window. The following subsections detail the RCC mechanisms.

4.1. The window assignment

In RCC, the window assignment algorithm (shown in Algorithm 1) is to assign suitable window size for each flow when plenty of flows are transmitted simultaneously into the same NIC of receiver. Algorithm 1 uses *queue_maxsize* to represent the maximum size of switch queue. The algorithm is to partition *queue_maxsize* into appropriate window sizes for each flow.

In the algorithm, *info* is a global array storing all the flow information while *use_flow_priority* reveals whether to use flow priority. The array *info* is initialized and updated when receiver receives a Synchronous (SYN) packet and a Finish (FIN) packet, as shown in Fig. 4. *use_flow_priority* indicates whether the packet has flow size information. If the received packets carry the information of flow size, RCC will use them as flow priority. Or else, RCC sets the

[3] In Fig. 10 of [13], the state machine at receiver works as follows. If the state machine has state CE = 0, when an ECN-marked packet arrives at receiver, it will send ACK ECN = 0 instead of ECN = 1 to sender and change the state to CE = 1. The state CE = 1 has similar behaviors.



(a) Delayed ACK time = 1. (b) Delayed ACK times = 3.

Fig. 3. Motivation of receiver potential.

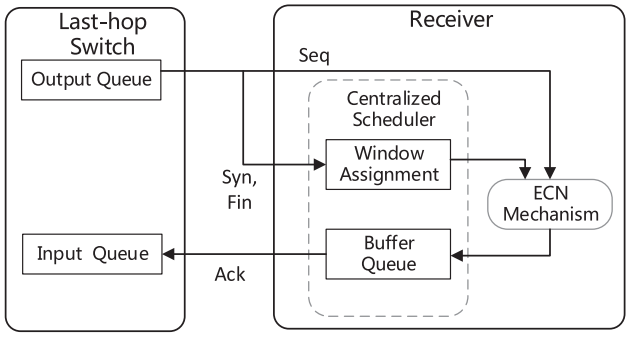


Fig. 4. RCC framework.

Algorithm 1 Assigning Windows for All Flows.

Require: *info //A global array storing all the flow information use_flow_priority //A flag to use flow priority;*

1: *Ascend*(*info*, *use_flow_priority*); $i = 0$; *sum_win* $= 0$;
2: **while** $i <$ *info.length*() **do**
3: $info[i].win = info[i].weight * queue\_maxsize$;
4: **if** $info[i].win < 1$ **then** $info[i].win = 1$;
5: **else if** *use_flow_priority* **and** $info[i].win > info[i].flowsize$ **then**
6: $info[i].win = info[i].flowsize$;
7: **end if**
8: $sum\_win += info[i++].win$;
9: **end while**
10: $i = info.length() - 1$; *N_noneed_check* $= 0$;
11: **while** *use_flow_priority* **and** *sum_win* $<$ *queue_maxsize* **do**
12: **if** $|\ info[i].win - info[i].flowsize\ | <= 1$ **then**
13: **if** $++N\_noneed\_check >= info.length()$ **then** *break*;
14: **end if**
15: **else** $info[i].win++$; sum_win++;
16: **end if**
17: **if** $--i == -1$ **then** $i = info.length() - 1$; *N_noneed_check* $=$ 0;
18: **end if**
19: **end while**

priority as zero. In Section 4.2, we show that when all flow priorities are the same or unavailable (i.e., $use\_flow\_priority = 0$), the buffer queue will degenerate into a First In First Out (FIFO) queue. In Section 6.3.1, we evaluate the performance when all flows have no priority.

In Line 1 of algorithm, *Ascend* is to sort flows by priority from high priority to low priority and calculate the flow weight. Suppose flow i's priority is p_i and there are n connections in total, then flow i's weight $w_i = p_i / \sum_j^n p_j$.

From Line 2 to 9, the **while** loop assigns congestion windows for all the incoming flows. For flow i, its window is $W_i = w_i \times queue\_maxsize$, which is an integer and takes packet as unit (Line 3). If W_i is smaller than 1, RCC assigns 1 to W_i to avoid stopping flows (Line 4). If W_i is larger than flow size, set W_i as flow size to avoid waste of bandwidth (Line 6). *sum_win* is used to utilize the left bandwidth in the next **while** loop (Line 8).

From Line 11 to 19, the **while** loop tries to utilize the left bandwidth unused by Lines 2 to 9. If the window size is close to flow size, this flow has no need to increase window (Line 13). In general, flow window is increased by one (Line 15). The loop will continue to check all the flows to utilize the left bandwidth until there is no bandwidth left or all flows are already assigned sufficient windows (Line 17).

When the number of connections is too large, it is possible for the sum of congestion windows to exceed *queue_maxsize*, e.g., when $queue\_maxsize = 40$, each of the 100 flows is assigned with one window, leading to 100 windows in total. Accordingly, Algorithm 1 does not suffice suppressing congestion. We assign at least one window to each flow to prevent starving TCP flow, which is important to long flows. RCC needs a complementary mechanism to absorb redundant windows which is shown in the next section.

4.2. The buffer queue

RCC has a buffer queue to store ACK packets temporarily. It assures the sum of flight windows is 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 arrives at receiver, the counter is reduced by packet size. On the contrary, when a new ACK is sent back, the counter will be increased by the increment of ACK-number.[4]

The centralized scheduler calculates the sum, *flight_wins*, of sent and unacknowledged windows from all the flows within the same NIC. When *flight_wins* is larger than *queue_maxs* − *ize*, the buffer queue will queue up ACK packets instead of sending ACKs immediately until *flight_wins* is below *queue_maxsize*. Besides, the buffer queue uses quicksort algorithm to sort the ACK packets in it by flow priority.[5] As described in Section 4.1, RCC uses flow size

[4] There are two variables for counting flight windows. One is used to record increment of ACK number every round, the other is the expected maximum sequence number. The details are ignored for brevity.

[5] The number of ACKs is not large because the number of concurrent senders is not large. Hence the sort algorithm does not consume large time and space. The main delay in buffer queue is analyzed in Section 5.1.

in packet as flow priority. The ACKs with the highest priorities are stored in the head of the queue.

The algorithm of buffer queue is presented in Algorithm 2. Once an ACK is sent back, the algorithm will be invoked and handle the ACK packet as input. Since each flow needs a distinct identifier, the buffer queue stores the thread pointer of flow.

Algorithm 2 Sending Back ACKs via Buffer Queue.

Require: *flags //The packet flags to be written in TCP packet header; flow_identity //The identifier of this flow; flow_priority //The priority of this flow;skiptime //Times that current ACK is skipped over;*

1: $skiptime = queue.erase(flow\_identity)$;
2: $queue.insert(flow\_identity, flow\_priority, flags, skiptime)$;
3: **while** $flight\_wins \leq queue\_maxsize$ **do**
4: $Send\ (queue.pop())$;
5: $Update\ (flight\_wins)$;
6: **end while**

Line 1 assures that the queue has a unique ACK corresponding to the current connection and returns *skiptime* which denotes the times that an ACK is sipped over (i.e., making no progress in buffer queues). Line 2 inserts the current-flow ACK into the queue. The inserting algorithm employs flow priority if necessary. However, the queue will degenerate into a 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. The maximum *skipptime* is $C \times RTO^r/2$, where C is bandwidth in packet per second, RTO^r is Retransmission TimeOuts for receiver in second and 1/2 is expectation of existence for higher priority.

Lines 3 and 6 send ACKs once there is idle space on the switch. The buffer queue plays a vital role in absorbing burtiness. Algorithm 2 is invoked when an ACK is generated by receiver. Previous techniques regarding pacing ACK packet such as RTT-related techniques are orthogonal and complementary to RCC by introducing them into Algorithm 2.

The centralized scheduler is targeted to eliminate congestion at the last-hop switch. In data centers, however, congestion occurs at other switches such as first-hop switches. To solve this problem, ECN is deployed at receiver.

4.3. Basic congestion control

As described earlier in Section 4, receiver communicates with sender by the receive-window size *rwnd* of advertisement window in ACK packet. After controlling sender to send data, we start deploying the receiver congestion control. First, receiver has to achieve basic congestion control such as TCP NewReno. Second, receiver has to estimate Round Trip Time (RTT), which is of significance for ECN to calculate congestion degree, α_r, and update window every RTT. Finally, the retransmission mechanism for ACKs may differ from TCP retransmission. This subsection shows basic congestion control and the following subsections present other aspects.

In RCC, the centralized scheduler targets congestion at last-hop switches while the ECN with basic congestion control aims at dealing with congestion far from receiver. Thus, the former assigns window size W_i while the latter adjusts window sizes in the range of 1 and W_i.

The basic congestion control of receiver in RCC is similar with TCP NewReno. It includes three parts that are Fast Recovery, Slow Start and Congestion Avoidance. RCC uses *seq* to indicate the sequence number of packet just arrived at the receiver, and uses *expected ack* to indicate the sequence number of data that the receiver is waiting to receive. The details are similar with TCP NewReno and skipped due to space limitation.

The basic congestion control has only a few codes to be realized. The only modification at sender is to use *rwnd* instead of its congestion window to send data.

4.4. RTT estimator

RCC ECN has to update congestion window every RTT. For TCP, after sender sends data packet of *seq*, the sender will estimate RTT when ACK $seq + segment$ arrives at the sender.[6] For RCC, after receiver sends ACK packet *expected ack*, the receiver will estimate RTT when data packet *expected seq* arrives at receiver. The following shows how to calculate *expected seq*:

$$expected\ seq = expected\ ack + rwnd - segment. \quad (2)$$

Once the receiver buffer queue sends off an *expected ack* to sender, it records *expected ack* and its sending time as $T1$ in a queue $q_{history}$. Next, when a *new seq* arrives at receiver at time $T2$, the receiver iterates $q_{history}$ to find whether there is an ACK whose *expected seq* is equal to *new seq*. If the receiver finds an ACK whose *expected seq* is equal to *new seq* and the ACK time is $T1$, the receiver will estimate RTT by $T2 - T1$ and delete this ACK from $q_{history}$. The arrived sequence number that is unequal to any *expected seq* in $q_{history}$ will not be estimated. The ACK that has waited in $q_{history}$ for a time larger than RTO^r will be deleted.

Hence, if there is no data packet arriving in receiver, the $q_{histroy}$ will finally be empty and no RTT will be estimated. Meanwhile, the above RTT measurement does not include the delayed time in $q_{history}$.

The RCC receiver starts estimating earlier than the sender, i.e., in the process of TCP 3-Way Handshake. Further, it estimates RTT only in states of Slow Start and Congestion Avoidance.

4.5. Delayed ACK and retransmission

In RCC, the receiver disables the delayed ACK mechanism (i.e., sending ACK every packet).

In addition, RCC leverages receiver's retransmission for ACK and uses RTO^r to clock ACKs.[7] The retransmission timeout is scheduled to fire after not receiving any packets or no corresponding ACK in the buffer queue for a time period of RTO^r. Another difference of retransmission between the sender and the receiver lies in line 7 of Algorithm 3. The receiver adds the PSH flag in the TCP header

Algorithm 3 Retransmission at Receiver.

Require: *flag //The flag of TCP header.*

1: $fast\ recovery\ flag \leftarrow 0$
2: $duplicated\ Ack \leftarrow 0$
3: $ssthresh \leftarrow \max\{2 * segment, rwnd/2\}$
4: $rwnd \leftarrow segment$
5: $delayAckCount \leftarrow delayAckMax - 1$
6: $RTO^r \leftarrow RTO^r * 2$
7: $SendACK(flag|PSH)$

[6] Assume *segment* is the maximum size of TCP packet and TCP packets are in general of *segment* size.

[7] Although 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 the receiver's control over congestion behaviors.

to force the sender to send the packet corresponding to the ACK immediately.

Also note that *SendACK* invokes Algorithm 2, and its *skiptime* will be added $C \times RTO^r/4$ due to flag PSH.

4.6. ECN response

In this subsection, we present ECN in RCC, which leverages ECN to detect the extent of remote congestion and avoid packet loss.[8] The ECN-enabled switch marks all the packets by setting the Congestion Experienced (CE) codepoint [34] when the queue length exceeds a certain threshold K. The RCC receiver detects the congestion extent 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:

$$rwnd = rwnd \times (1 - \alpha_r/2). \tag{3}$$

The receiver is able to directly get α_r and adjust congestion window. Hence, it does not need a dual mode state-machine to determine whether to set ECN-Echo bits.

5. Analysis of RCC

This section further analyses the parameters of centralized scheduler, ECN mechanism and RCC FCT.

5.1. Analysis of centralized scheduler

This section aims to analyze the queue size of last-hop switch and the buffer queue size of the RCC receiver. Suppose that ECN is disabled, and N long flows flood into the same NIC at receiver whose threshold is *queue_maxsize* in unit of packet. The average round-trip time without any queue backlog is RTT_a. The link capacity is C packet per second. Then the output queue size at the last-hop switch is

$$Q_1 = queue\_maxsize - RTT_a \times C. \tag{4}$$

Further, the average RTT at the stable state is

$$Q_1/C + RTT_a. \tag{5}$$

If N is smaller than *queue_maxsize*, the size of buffer queue is zero. Otherwise, the queue size is

$$Q_2 = N - queue\_maxsize. \tag{6}$$

Q_2 takes ACK packet as unit. However, since the output behavior of buffer queue is inspired by the input of received data packet, its queue delay is still Q_2/C. Further, the ideal maximum buffer queue size is

$$Q_{2\_ideal} = RTO^r \times C. \tag{7}$$

When the buffer queue size is greater than Q_{2_ideal}, it will induce spurious retransmission at receiver, which is unavoidable when N is large. However, this retransmission has little impact on RCC, because the retransmitted ACK is still queued in the buffer queue, subjecting to the centralized scheduler. In Fig. 5(a), the Q_1 is around 40 packets which is in line with Eq. (4), and we find Q_2 is in line with Eq. (6).

5.2. Analysis of ECN

This section focuses on the amplitude of queue oscillations (A_r) with ECN. Suppose the centralized scheduler is disabled. N long-lived flows with same *RTT* share a single bottleneck link of capacity C. N flows are synchronized, which always occurs in data centers and in wide area networks. Let $P(W\_1, W\_2)$ be the number of packets sent by the sender when window size increases from W_1 to $W\_2 > W\_1$. Since this takes $W\_2 - W\_1$ round-trips, during which the average window size is $(W\_1 + W\_2)/2$,

$$P(W\_1, W\_2) = (W\_2^2 - W\_1^2)/2. \tag{8}$$

When the switch queue size reaches K, the window size W_k of each flow will be

$$(C \times RTT + K)/N. \tag{9}$$

Next, the receiver reacts to these ECN marks after *RTT*/2, during which its window size increases by 1/2 packet, reaching $W\_k + 1/2$. The fraction of marked packets, α_r, is

$$\frac{P(W\_k, W\_k + 1/2)}{P((W\_k + 1/2)(1 - \alpha_r/2), W\_k + 1/2)} \tag{10}$$

Plugging (8) into (10), and assuming $W\_k >> 1$, we have

$$\alpha_r^2(1 - \alpha_r/4) = \frac{W\_k + 1/4}{(W\_k + 1/2)^2} \approx \frac{1}{W\_k}.$$

Assume α_r is small, so

$$\alpha_r \approx \sqrt{\frac{1}{W\_k}}. \tag{11}$$

Hence, the oscillation amplitude in window size of a single flow, D, is given by

$$\begin{aligned} D &= (W\_k + 1/2) - (W\_k + 1/2)(1 - \alpha_r/2) \\ &= (W\_k + 1/2)\alpha_r/2. \end{aligned} \tag{12}$$

Since there are N flows in total,

$$A_r = ND = N(W\_k + 1/2)\alpha_r/2. \tag{13}$$

Plugging the value of Equations (9) and (11) and $W\_k >> 1$, we have

$$A_r = 1/2\sqrt{N(C \times RTT + K)}. \tag{14}$$

We further offer the period of oscillations (T_c) as follow,

$$T_c = D = 1/2\sqrt{(C \times RTT + K)/N}\ (in\ RTTs). \tag{15}$$

And the maximum queue size (Q_{max}) is

$$Q_{max} = N(W\_k + 1/2) - C \times RTT = K + N/2. \tag{16}$$

Since

$$\begin{aligned} Q_{min} &= Q_{max} - A_r \\ &= K + N/2 - 1/2\sqrt{N(C \times RTT + K)}, \end{aligned} \tag{17}$$

to find a lower bound on K, we minimize (17) over N and choose K so that this minimum is larger than zero. Then we get

$$K > (C \times RTT)/7. \tag{18}$$

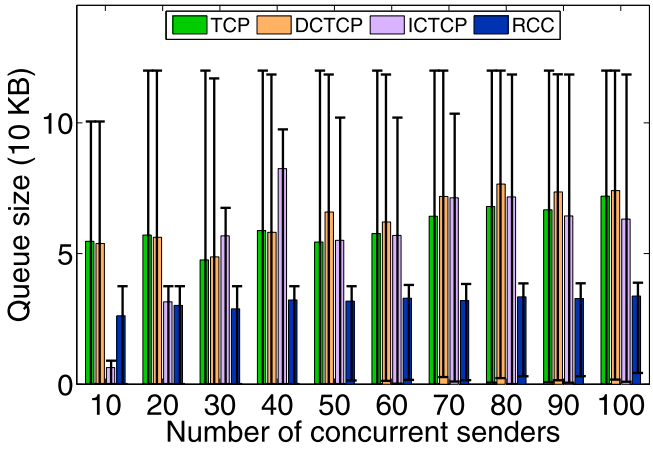
Eq. (14) reveals that when N is small, the queue oscillation amplitude of RCC is in $O(\sqrt{C \times RTT})$. In Fig. 5(a), A_r is above 4 packets and T_c is around 1 RTT, both of which are in line with Equations (14) and (15).

5.3. Analysis of RCC near-optimal FCT in incast

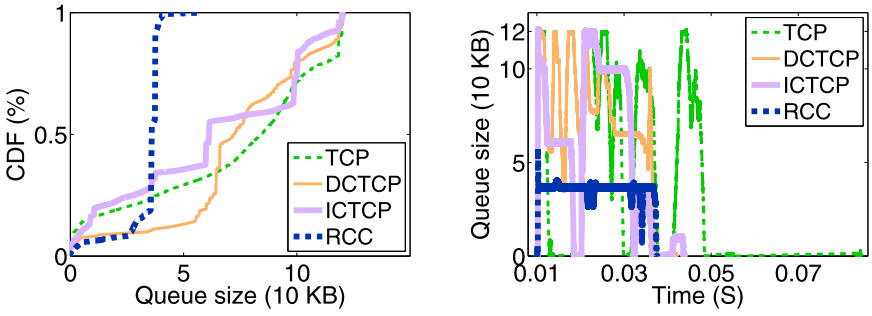
To analyze RCC near-optimal FCT in incast, this section assumes there is a near-Optimal incast TCP protocol (OITCP) who has all flow information in advance. For simplicity, this section omits the link delay and assumes that all the hosts are under the same switch forming a many-to-one traffic pattern, where senders send flows of the same size to one receiver.

The switch queue size is Q in packet and the receiver link capacity is C packet per second. Denote the number of flows by N and the flow size by L in byte. And *queue_maxsize* and Q_o denote

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



(a) The mean size with the 5th and 95th percentile of queue size.



(b) The CDF of queue size with 100 senders. (c) Time series of queue size with 100 senders.

Fig. 5. Queue buildup in the last-hop switch.

the switch queue sizes in packet for RCC and OITCP at the steady state, respectively.

Meanwhile, OITCP aims to maintain a switch queue size of Q_o. It is able to transmit window of arbitrary size and partition $Q_o \times segment$ averagely among flows.

Similar to large-scale incast, suppose

$$queue\_maxsize \leq N \leq Q_o, \tag{19}$$

such that OITCP flows will get a window size more than one byte by $Q_o \times segment/N$ and RCC flows will get a window size of one *segment*.

Hence, an OITCP flow will transmit

$$L/(Q_o \times segment/N) = NL/(Q_o \times segment) \tag{20}$$

packets. OITCP experiences a switch queue delay Q_o/C and a transmission delay $L/(segment \times C)$. Finally, its FCT is

$$(NL/(Q_o \times segment)) \times (Q_o/C) + L/(segment \times C) = \frac{NL+L}{segment \times C}. \tag{21}$$

On the other hand, an RCC flow will transmit $L/segment$ packets and experience two queue delays and a transmission delay $L/(segment \times C)$. The two queue sizes Q_1 and Q_2 are presented in Section 5.1. Hence the total queue delay is

$$\frac{queue\_maxsize}{C} + \frac{N - queue\_maxsize}{C} = N/C. \tag{22}$$

Thus, its FCT is

$$(L/segment) \times (N/C) + L/(segment \times C) = \frac{NL+L}{segment \times C}, \tag{23}$$

which is the same as the near-optimal FCT in Eq. (21). This proves that in an ideal environment, RCC is nearly optimum.

6. Evaluation

This section evaluates RCC. Section 6.1 examines RCC in terms of queue buildup, goodput, packet-drops and FCT in heavy incast. Section 6.2 investigates the performance of RCC in at-scale simulations. Section 6.3 provides transport performance evaluations regarding to flow priority, dynamic traffic, multiple bottleneck topology, as well as convergence and coexistence with TCP.

In ns3, link delay is 40 μs and bandwidth is 1 Gbps. The size of switch output queue is 128 KB (i.e., 80 1500-B packets). All the transport mechanisms are with the delayed ACK mechanism disabled, thus the passive effect of dual state machine of DCTCP is limited. The Retransmission TimeOuts (i.e., RTO^s for sender and RTO^r for receiver) are set to 10 ms, which is in line with that in [7]. The ECN marking threshold K is set to 40 packet, and *queue_maxsize* is set to 40 packet. The switch queue adopts the DropTail mechanism. We use flow size as priority. All the parameters remain unchanged in the following subsections unless otherwise noted.

6.1. Heavy incast

In heavy incast, concurrent senders start to transmit 32-KB flows to one receiver at the same time, which is explained in Section 3.1. All the hosts are under the same switch.

6.1.1. Queue buildup

The mean of queue size in the last-hop switch with four transport mechanisms is shown in Fig. 5(a). RCC 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% decrease in the mean queue size over TCP, DCTCP and ICTCP, respectively.

No matter how many senders there are, RCC maintains a stable mean queue sizes. In addition, the 95th-percentile queue sizes of RCC hold low and stable. These results come from the centralized scheduler of RCC. Meanwhile, the buffer queue at the RCC receiver has limited the flight windows on the path to a smaller size than that of a switch queue. Consequently, RCC has the slightest queue fluctuation in all cases.

To observe more details of queue buildups, Fig. 5(b) and (c) demonstrate the Cumulative Distribution Function (CDF) and the instant size of queue length in the 100-sender scenario of incast. In Fig. 5(b), RCC maintains lower queue length in the majority of time and never reaches the maximum queue size while other protocols suffer from unstable queue and overflow.

In Fig. 5(c), RCC has the minimum and stablest queue size most of the time. This is attributed to its centralized scheduler which guarantees that *flight_wins* is below the *queue_maxsize* of 60 KB. In Fig. 5(c), the queue size of RCC is around 40 KB in the steady state. This is because the basic RTT is 160 μs and the bandwidth is 1 Gbps, producing 20 KB difference between 60 KB and 40 KB. There are two decreases with RCC because higher priority flows have finished and their CLOSE packets are delayed on the path to the receiver, resulting in lacking enough packets on the path. However, they have the minimum impact on queue size. At the beginning of transmission, the SYN packets coexist with data packets, thus leading to a larger queue size in RCC. In Fig. 5(c), other protocols experience severe queue oscillations.

6.1.2. Goodput and packet-drops

As revealed in Fig. 6(a), all the mechanisms except RCC suffer from goodput collapse in incast. With the number of senders increasing, RCC achieves better goodput among mechanisms, which benefits from its zero packet-drops illustrated in Fig. 6(b). 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. For ICTCP, its behavior is similar to Fig. 10 in [14] when sender number is below 50. With the number of senders larger than 50, ICTCP suffers from goodput collapse. This is because its control interval 2*RTT is too long to suit prompt switch queue increasing. Taking 50 senders for example, at the second RTT, there are 100 packets sent by 50 senders, leading to overflow of switch queue of 80-packet size.

As shown in Fig. 6(b) with logarithmic *y*-axis, RCC does not drop packets in all cases because of its buffer queue absorbing burstness. DCTCP drops less packets than TCP and ICTCP due to effective ECN. ICTCP leverages throughput differences to respond to congestion, thus dropping less packets than TCP.

Fig. 6(c) depicts ECN marking times with logarithmic *y*-axis. Note that TCP and ICTCP have no ECN mechanisms to cope with ECN packets. In Fig. 6(c), the numbers of markings are always 0 for RCC while other protocols have hundreds of marking times. This because RCC maintains a stable and low size in switch queue, as illustrated in Fig. 5(a).

6.1.3. Flow completion time

This subsection presents FCT of RCC in different heavy incast scenarios. Fig. 7(a) plots the 1st, 50th and 99th percentile FCTs of four mechanisms with varying number of senders, each of which sends 32 KB flow to the same receiver. As portrayed in Fig. 7(a), with the growing number of senders, all the 50th-percentile FCTs increase. RCC's 99th FCTs holds the lowest, which is vital in partition/aggregator construction where the last flow determines QoS. RCC has 51.2%, 16.4% and 11.7% mean decrease in the 99th-percentile FCT over TCP, DCTCP and ICTCP, respectively. Beyond that, RCC has the smallest difference between the 1st and 99th percentile FCTs (except the case of 30 senders), implicating that RCC has stable performance.

Fig. 7(c) plots CDFs of FCTs with 90 senders. RCC has two clear FCT distributions. This character is attributed to its buffer queue which queues up ACK packets by flow priority. When 90 flows with the same flow size (i.e., same priority) arrive at receiver, only 40 flows continue to transmit while other flows have to wait in the buffer queue temporarily. Therefore, the FCT of RCC is divided into two parts. The first part has around 20-ms FCTs while the second part has around 33-ms FCTs. ICTCP, DCTCP and TCP have three, four and five evident FCT distributions, respectively.

In Fig. 7(b) and (d), the number of senders is fixed to 30. The 1st, 50th and 99th percentile FCTs with varying flow sizes are illustrated in Fig. 7(b) while the CDF of FCTs with 30 senders sending 800-KB flows are plotted in Fig. 7(d).

In Fig. 7(b), RCC has the minimum 99th-percentile FCTs. Besides, it achieves the second smallest 1st-percentile FCTs. Although TCP achieves the minimum 1st-percentile FCTs, its 99th-percentile FCTs are still large due to severe packet-drops caused by its proactive congestion control. In Fig. 7(b), both DCTCP and ICTCP have achieved focused FCT distributions because 30 senders cause a moderate incast which DCTCP and ICTCP are effective to solve. In Fig. 7(d), RCC has two clearer FCT distributions due to the same reason in Fig. 7(c). Owing to effective congestion control, DCTCP and ICTCP achieve comparable FCTs with RCC.

The next subsection further testifies the RCC performance in a typical datacenter network.

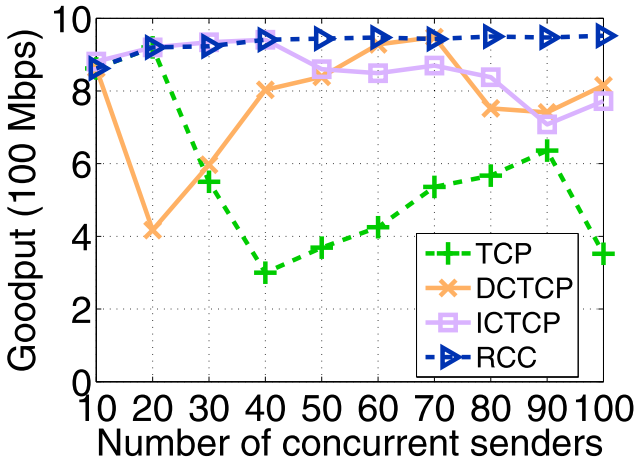
6.2. At-scale simulations

In this subsection, the datacenter network has a typical fat-tree topology in which a root switch node is adopted to connect all the Top of Rack (ToR) switches to form a core network. The ToR servers are connected via 1-Gbps links, and the ToR switches connect to the core switch via 1× number-of-servers-under-a-ToR Gbps. The number of servers under a ToR network is 25 and the number of ToR networks is 40. Among the 1000 hosts, 5 random receivers is chosen as aggregators. Short-flows comply with the Pareto distribution with mean 50 KB and shape 1.2. The arrival times of flows comply with exponential distributions with means of 300 μs, 400 μs, 500 μs, 600 μs and 700 μs. Meanwhile, there are 5 long-lived flows from 5 random senders as background traffic, each of which is 30 MB and sent to one of the 5 receivers, occupying 75 percent of traffic in total in datacenter networks.

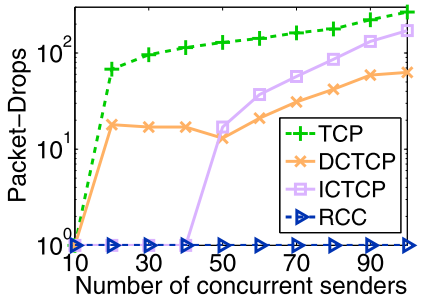
The offered load of datacenter network ranges from 0% to 90%. The results are illustrated in Fig. 8 showing short-flow FCT percentiles (Fig. 8(a)), long-flow throughput (Fig. 8(b)) and packet-drop times in ToR switches (Fig. 8(c)) and the core switch (Fig. 8(d)). The *y*-axis in Fig. 8(a) is logarithmic in 100-ms unit.

In Fig. 8(a), RCC achieves the minimum 95th- and 99th-percentile FCTs (except for the case of 40 senders with a slightly higher 99th-percentile FCT). With heavy load (greater than 50% of the offered load), RCC still keeps low FCTs due to its effective centralized scheduler. RCC also has small gaps between the 1st and 99th percentile, which indicates its stable and predictable performance. On the other hand, other protocols experience long-tail FCTs in heavy-load networks, because congestion control fails in the face of heavy traffic.

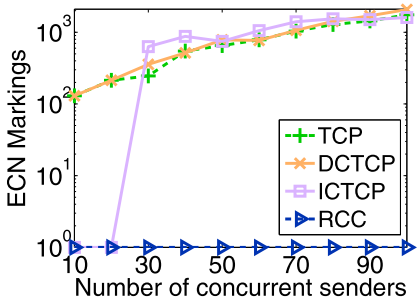
In Fig. 8(b), the lowest, median and highest throughput of five 30-MB flows of four mechanisms in varying offered load are plot-



(a) Goodput.

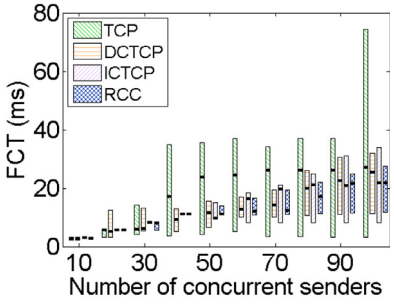


(b) Packet-drop times.

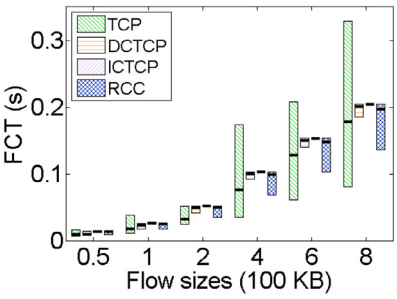


(c) ECN markings.

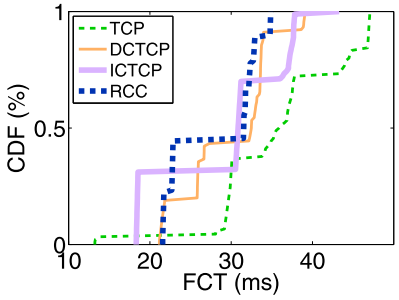
Fig. 6. Goodput and the number of packet-drops and markings of packet in the last-hop switch queue in incast.



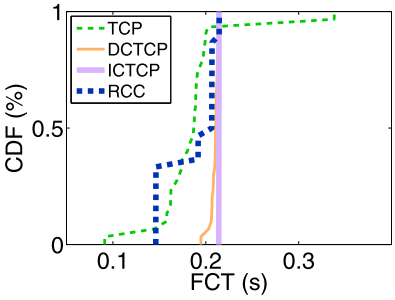
(a) The 1st, 50th, 99th of FCT with varying number of senders sending 32 KB flows.



(b) The 1st, 50th, 99th of FCT with 30 senders sending flows of varying sizes.

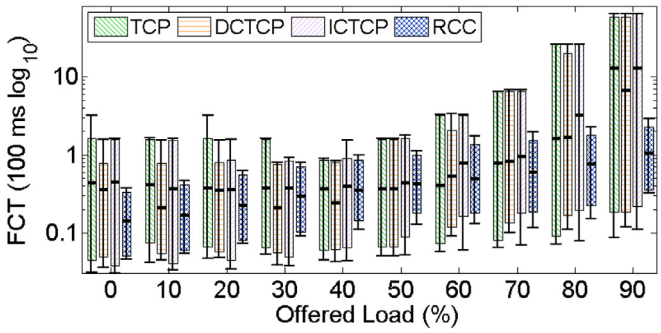


(c) 90 senders with 32 KB each.

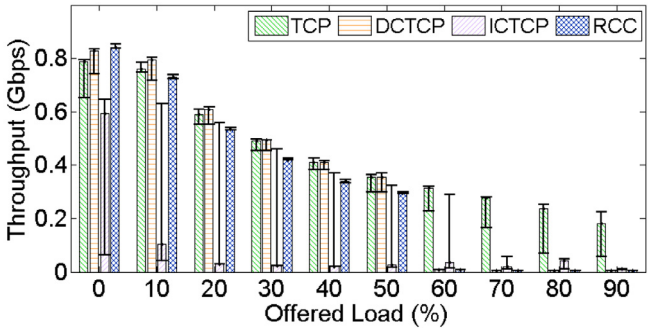


(d) 30 senders with 800 KB each.

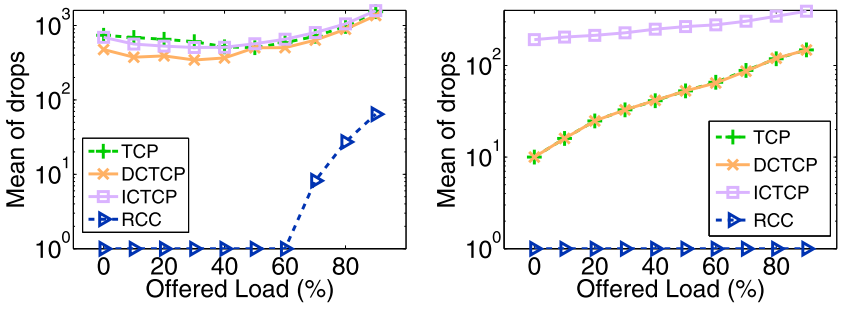
Fig. 7. FCTs with varying sender numbers and flow sizes.



(a) The 1st, 5th, 50th, 95th and 99th percentile of short-flow FCTs.



(b) The lowest, median and highest of throughput of long flows.



(c) Mean pkt-drops in last-hop ToRs. (d) Mean pkt-drops in the root switch.

Fig. 8. A typical data center with varying offered load.

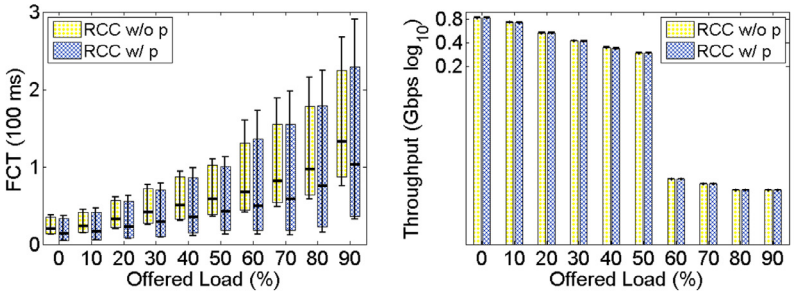
ted. In the figure, in light-load networks (no greater than 50% of the offered load), RCC has moderate preference for long flows due to its prudent scheduler that aims to keep the switch queue short, hence hurting throughput. Also notable is with heavy load (no less than 60% of the offered load), the RCC throughput decreases drastically. This is because 60% offered load has already congested the switch queues, which triggers ECN marking whose marking threshold is 40 packets. In addition to absorbing burstness, RCC further reduces congestion window after receiving ECN-marked packets, leading to a sharp decrease on long flow throughput.

Moreover, RCC reveals the minimum difference between the lowest and highest throughput, because 5 long flows have the same priorities. ICTCP has large gaps between the highest and middle throughput, implicating its bias against long flows. TCP and DCTCP have higher throughput at the cost of large queue size and packet-drops.

All the drop issues can be divided into two parts. The first part is the majority of drops, which occurs in the output queues at the last-hop switches of 5 receivers, as shown in Fig. 8(c). The second part occurs in the output queues at the root switch connected to the last-hop switches, as shown in Fig. 8(d). The two figures illustrate the average drops of the 5 output queues corresponding to 5 receivers.

In Fig. 8(c), RCC starts to drop packets when the offered load is of 60%. The average packet-drops of RCC are 1, 8, 27 and 64 in 60%, 70%, 80% and 90% offered load, respectively. The fewest drops are attributed to its centralized scheduler which aims to eliminate the last-hop switch congestion. On the contrary, other protocols experience severe packet-drops. DCTCP has less drops than TCP and ICTCP because its ECN effectively responds to congestion, implicating that ECN is more effective than the throughput-detection mechanism of ICTCP in avoiding packet loss. In Fig. 8(d), RCC does not drop packets. The reason is that its centralized scheduler effectively suppresses congestion in the last-hop switches, resulting in light congestion at the root switch.

Meanwhile, Fig. 8(d) reveals two points worth noting. First, TCP and DCTCP have the same drops, implicating that ECN has no effect on congestion that is far away from receiver. This is because



(a) The 1st, 5th, 50th, 95th and 99th percentile of short-flow FCTs.

(b) The lowest, median and highest long-flow throughput.

Fig. 9. FCTs and throughput with and without flow priority.

the severe congestion on the last-hop switches covers the congestion information of the root switch queues. Second, ICTCP still has hundreds of drops at root switch queues, implicating that ICTCP is blunt to congestion far from receiver.

6.3. Extension evaluations

6.3.1. Without flow priority

This subsection examines the performance of RCC when flows have no flow priority. The other configurations are kept the same with that in Section 6.2. As described in Sections 4.1 and 4.2, without flow priority, the window assignment partitions *queue_maxsize* averagely and the buffer queue becomes FIFO. In Fig. 9(a) and (b), FCTs and throughput of RCC are offered with and without flow priority, respectively.

Counter-intuitively, in Fig. 9(a), the 99th-percentile FCTs is lower without priority. But the 1st-, 5th- and 50th-percentile FCTs increase. We attribute this to the FIFO queue which does not favor any flows, leading to a more centralized FCTs. In Fig. 9(b), RCC without priority achieves higher throughput than RCC with priority when the offered load is less than 50%, and it reveals the same throughput with RCC with priority when the offered load is no less than 60%. The minute differences (no greater than 10 Mbps) is because that there are many short flows overwhelming the minority of long flows at receiver. Generally speaking, RCC still works well without flow priority.

6.3.2. Robustness to burstiness

This subsection examines RCC for robustness to burstiness which is common in datacenter networks. To show burstiness, all the hosts are placed under the same switch, and the relevant configurations are the same as in Section 6. A long-lived flow of 10 MB is transported as background traffic starting at 0 s and 60 50-KB short flows start at 0.01 s. Four mechanisms are leveraged and the results of throughput, queue size, FCTs, packet-drops and goodput are illustrated in Fig. 10.

Fig. 10(a) shows long-flow throughput of four mechanisms in burstiness. The RCC throughput responds to burstiness smoothly and timely. Its centralized scheduler responds to burstiness in TCP 3-Way Handshake of short flows, resulting in quick and reasonable response. When all the short flows finish transmitting, RCC re-assign congestion windows to the long flow, making full use of bandwidth. DCTCP and TCP throughputs fluctuate during burstiness, indicating that they are less sensitive to burstiness. After 0.05 s, both TCP and ICTCP have lower throughput. The former is because TCP experiences packet loss during burstiness and enters congestion avoidance. The latter is because ICTCP uses the fixed Global Slot to allocate bandwidths. However, in the first subslot of the Global Slot, the windows remain unchanged, leading to idle links and lower throughput.

In Fig. 10(b), TCP, DCTCP and ICTCP have large queue size when burstiness occurs, leading to packet loss. And their queue oscillate, resulting in unstable performance. For RCC, it achieves the minimum and stable queue size when facing burtiness, implicating that it absorbs burstiness by buffer queue in time.

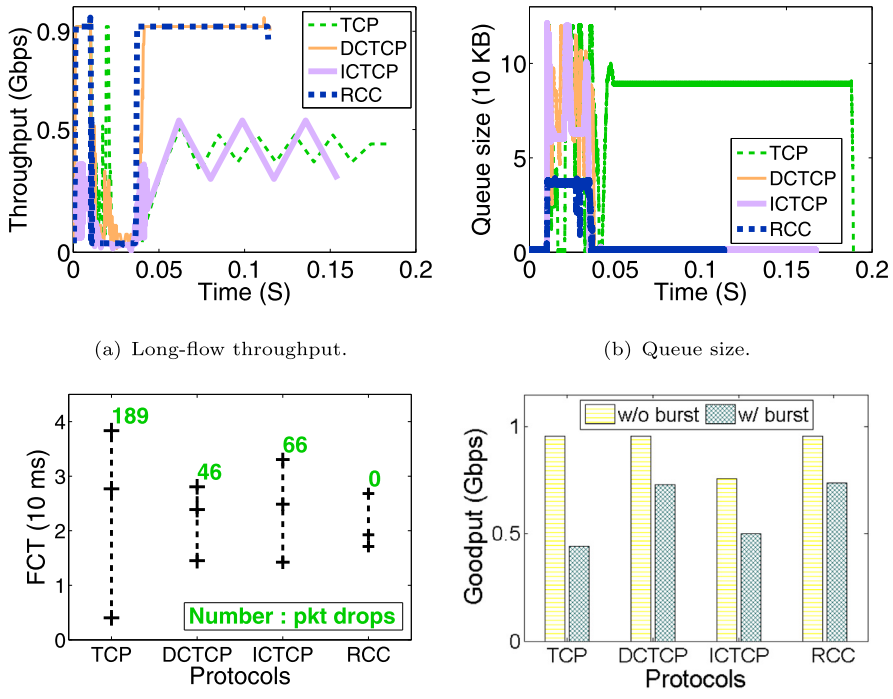
Fig. 10(c) offers the packet-drops and FCTs of short flows in burstiness. In the figure, TCP has the most packet drops while RCC does not drop packets on the switch queue. In addition, RCC has the lowest 50th- and 99th-percentile FCTs (i.e., 19 ms and 27 ms, respectively) at the cost of the highest 1st-percentile FCT, which suits datacenter networks where the 99th-percentile FCT is more important to than the 1st-percentile FCT. RCC sacrifices the minority of flow FCTs to satisfy most flow FCTs.

To analyze the impact of burstiness on long-flow goodput, we re-perform the scenario without burstiness and show long-flow goodputs with (the blue bar on the right) and without (the yellow bar on the left) burstiness for each mechanism in Fig. 10(d). ICTCP has lower goodput without burstiness. This is because ICTCP has windows unchanged in the first subslot, leading to an idle link and lower goodput. In the figure, all mechanisms have a downgraded goodput when encountering burstiness. RCC still achieves the highest goodput (i.e., 737 Mbps) in the face of burstiness due to its window assignment which re-allocates windows to the long flow once short flows finish.

6.3.3. Multiple bottleneck topology

To investigate the RCC performance in bottleneck topologies, this subsection emulates a dual bottleneck topology. As depicted in Fig. 11(a), 3 groups of senders send data to 2 groups of receivers. Groups S1 and S3 (each with 20 senders) both send long-lived (1-MB) flows to receiver R1, while each sender in group S2 sends long-lived (1-MB) flow to an assigned receiver in group R2. In such topology, two bottlenecks emerge. One lies between the *ToR-1* switch and the middle switch, while the other is between the *ToR-2* switch and receiver R1. Flows from S1 encounter both bottlenecks.

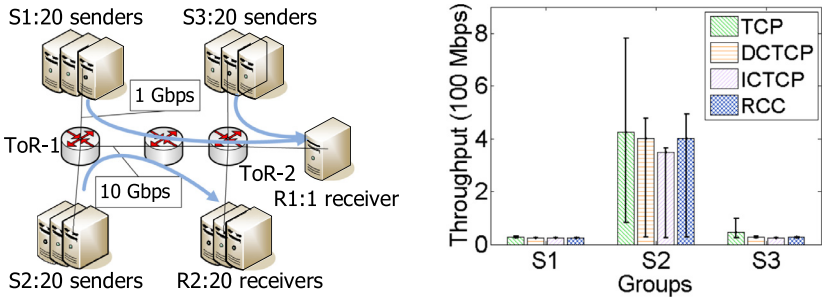
Four transport mechanisms are performed on the topology and the 1st, 50th and 99th percentile throughput of sender groups are shown in Fig. 11(b). With TCP, group S2 has the largest 99th and 50th percentile throughput (780 Mbps and 426 Mbps, respectively) while with RCC group S2 has the second-largest 99th throughput (494 Mbps). All mechanisms have small throughput in groups S1 and S3, because the flows from S1 and S3 are throttled by bottleneck at the *ToR-2* output queue. ICTCP behaves conservatively in congestion control, thus having the lowest 99th and 50th throughput in S2.



(a) Long-flow throughput. (b) Queue size.

(c) The 1st, 50th and 99th FCTs of short flows and pkt drops. (d) Long flow goodput with(o.) burstiness.

Fig. 10. Burstiness with 60 50-KB short flows and a 10-MB long flow.



(a) The topology of multi-bottleneck. (b) The 1st, 50th and 99th throughput of sender groups.

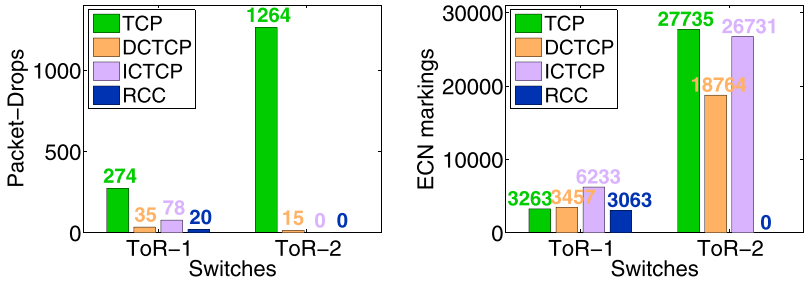
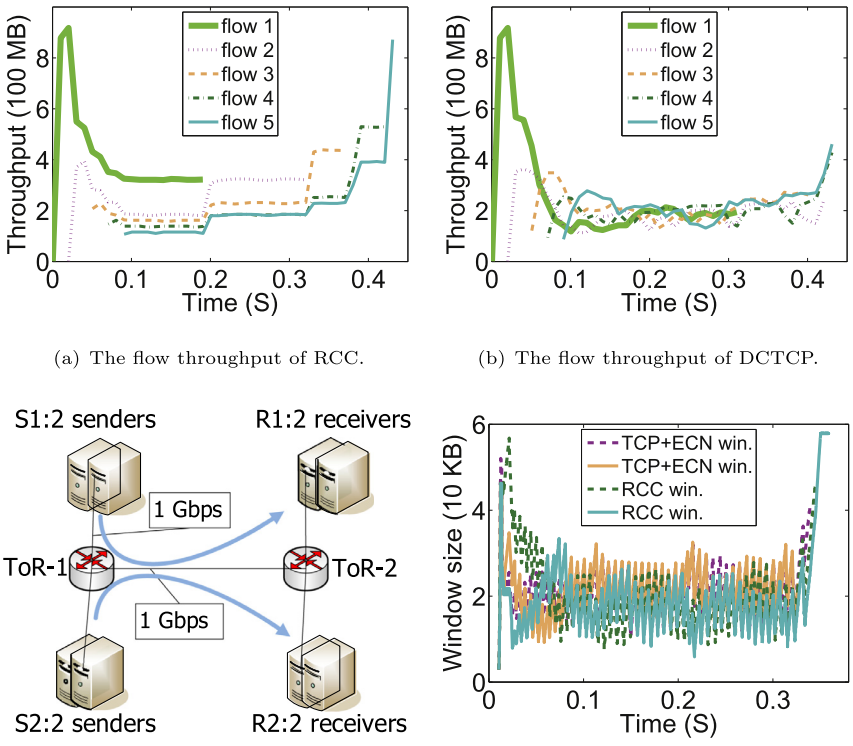


Fig. 11. Multiple-bottleneck topology for throughput experiment.

Although RCC achieves comparable throughput with other protocols, it suffers only 20 and zero packet-drop(s) in *ToR-1* and *ToR-2* respectively in Fig. 11(c). And RCC achieves the least marking times in Fig. 11(d), suggesting it achieves the higher network utilization among four mechanisms. In spite of the largest 99th and 50th throughput in all groups in Fig. 11(b), TCP has the most severe packet-drops in Fig. 11(c) and corresponding marking times in Fig. 11(d), indicating that TCP wastes network resources, e.g., bandwidth.

ICTCP has the second largest packet-drops in *ToR-1* in Fig. 11(c) and the largest marking times in *ToR-1* in Fig. 11(d), justifying ICTCP's inefficiency on coping with congestion far from receiver. In conclusion, while RCC does not achieve the highest throughput among three sender groups, it has the least packet-drops and



(a) The flow throughput of RCC.

(b) The flow throughput of DCTCP.

(c) The topology for coexistence test.

(d) The cong. win. of TCP + ECN and RCC flows.

Fig. 12. The throughput of five flows and the performance of RCC and TCP+ECN in bottleneck.

marking times in both *ToR-1* and *ToR-2* by virtue of its cooperative centralized scheduler and the ECN mechanism.

6.3.4. Fairness for flow with priority

To evaluate RCC in terms of fairness, this subsection connects 6 servers under the same switch via 1-Gbps link to form a 5-to-1 traffic pattern. Each of the 5 servers sends 10 MB flows to the same receiver step by step with an interval of 20 ms. RCC and DCTCP are performed separately and the throughput of RCC and DCTCP is shown in Fig. 12(a) and (b). In Fig. 12(a), RCC responds to the arrival of flows smoothly and proportionally, making full use of bandwidth. Note that RCC does not assign bandwidth evenly because it uses flow size as priority. And RCC never causes packet loss. In Fig. 12(b), DCTCP has somewhat equal bandwidth for each flows after arrivals of flows. In Fig. 12(a), RCC does not starve the flows with low priority as its buffer queue prevents lower priority from being waited for a long time.

To sum up, RCC has better fairness on flow priority by its centralized mechanism at receiver.

6.3.5. Coexistence with TCP

This subsection examines the window evolution of RCC when coexisting with TCP. To evaluate the RCC ability under such circumstance, this subsection uses the bottleneck topology shown in Fig. 12(c). In the topology, 2 senders in the S1 send 10-MB TCP flows to the corresponding 2 receivers in R1 while 2 senders in S2 send 10-MB flows to the corresponding receivers in R2 by RCC. The ToR switches have the ECN mechanism enabled.

In the scenario, the queue sizes in the bottleneck are always around 70 KB, which means that the ECN mechanism takes effect. In addition, Fig. 12(d) plots the evolution of congestion windows of four flows, two TCP flows and two RCC flows. From the figure, TCP flows have comparable window size with that of RCC flows, indicating that RCC coexists with TCP and ECN well. The ECN mechanism makes RCC have a moderate performance when coexisting with other protocols. If the ECN mechanism is disabled, RCC still keeps one window size, i.e., 1500 bytes in this paper, when facing other protocols.

7. Conclusion

This paper presents RCC, a congestion control mechanism at receiver, to solve heavy incast in datacenter networks that are expanding continuously. RCC is constructed by two main mechanisms, open- and closed-loop congestion control to cope with different congestions. First, the centralized scheduler that consists of window assignment and buffer queue is leveraged to cope with the congestion in the last-hop switch. Second, the ECN mechanism is deployed to cope with other congestions. The evaluation illustrates that RCC suits higher-degree many-to-one communication patterns and has effectively suppressed congestion.

This paper derives several interesting observations. First, RCC reveals the potential of receiver to deal with link congestion. Receiver is able to aggregate the information received over all incoming TCP flows and obtain more accurate congestion information than sender, as shown in Section 3.2. Second, lower queue size may not hurt FCT, as shown in Section 6.1.3, while higher throughput may sacrifice precious network resources, i.e., inducing packet loss, as illustrated in Section 6.2. Third, in datacenter networks, congestions occur in different places, e.g., the last-hop switch and the first-hop switch, making it necessary to design different solutions, as shown in Section 6.3.3. Fourth, efficient datacenter transport may not necessitate flow priority, as shown in Section 6.3.1.

RCC is presented in this paper in ns3 which has exact realization of real-world TCP. The centralized scheduler of RCC is simple and the main deployment cost is shifting congestion control from

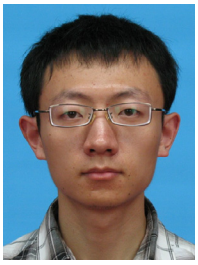
sender to receiver. The code line of RCC in ns3 is no more than 1, 000. RCC is the first attempt to improve congestion control by realizing congestion control of TCP-related protocols at receiver. It is challenging to extend RCC from ns3 to real-world, but RCC still shows possibility to improve TCP protocols.

Acknowledgments

We appreciate the help from Chi-Yao Hong and Balajee Vamanan in terms of technical realization. We also would like to thank the anonymous reviewers for valuable their time and suggestions. This work was supported by the National Natural Foundation of China (61170292, 61472212), National Science and Technology Major Project of China (2015ZX0300-3004), the National High Technology Research and Development Program of China (863 Program) (2013AA-013302, 2015AA015601), EU Marie Curie Actions CR-OWN (FP7-PEOPLE-2013-IRSES-610524).

References

[1] How big is facebooks data? 2.5 billion pieces of content and 500+ terabytes ingested every day, http://goo.gl/n8xhq.
[2] Microsoft accelerates its data center expansion, http://t.co/eZXrOQqQBu.
[3] Dropbox clears 1 billion file uploads per day, http://cnet.co/YYwgB3.
[4] Y. Cui, H. Wang, X. Cheng, B. Chen, Wireless data center networking, IEEE Wireless Commun. 18 (December 2011) 46–53.
[5] J. Dean, S. Ghemawat, Mapreduce: simplified data processing on large clusters, Commun. ACM 51 (January 2008) 107–113.
[6] M. Isard, M. Budiu, Y. Yu, A. Birrell, D. Fetterly, Dryad: distributed data-parallel programs from sequential building blocks, Proc. EuroSys. 41 (June 2007) 59–72.
[7] V. Vasudevan, A. Phanishayee, H. Shah, E. Krevat, D.G. Andersen, G.R. Ganger, G.A. Gibson, B. Mueller, Safe and effective fine-grained tcp retransmissions for datacenter communication, Proc. ACM SIGCOMM 39 (October 2009) 303–314.
[8] R. Griffith, Y. Chen, J. Liu, A. Joseph, R. Katz, Understanding tcp incast throughput collapse in datacenter networks, in: Proc. of ACM WREN, August 2009, pp. 73–82.
[9] J. Zhang, F. Ren, C. Lin, Modeling and understanding tcp incast in data center networks, in: Proceedings of IEEE INFOCOM, April 2011, pp. 1377–1385.
[10] N. Dukkipati, N. McKeown, Why flow completion time is the right metric for congestion control, ACM SIGCOMM Comput. Commun. Rev. 36 (January 2006) 59–62.
[11] J. Brutlag, Speed matters for google web search, http://code.google.com/speed/files/delayexp.pdf (2009).
[12] A. Sivaraman, K. Winstein, P. Thaker, H. Balakrishnan, An experimental study of the learnability of congestion control, in: Proceedings of ACM SIGCOMM, 2014, pp. 479–490.
[13] M. Alizadeh, A. Greenberg, D. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, M. Sridharan, Dctcp: efficient packet transport for the commoditized data center, in: Proceedings of ACM SIGCOMM, August 2010, pp. 63–74.
[14] H. Wu, Z. Feng, C. Guo, Y. Zhang, Ictcp: incast congestion control for tcp in data center networks, IEEE/ACM Trans. Networking (TON) 21 (April 2013) 345–358.
[15] L. Xu, K. Xu, Y. Jiang, F. Ren, H. Wang, Enhancing tcp incast congestion control over large-scale datacenter networks, in: Proceedings of IEEE IWQoS, June 2015, pp. 225–230.
[16] J. Postel, The newreno modification to tcp's fast recovery algorithm, September, RFC 793, 1981.
[17] W. Stevens, Tcp slow start, congestion avoidance, fast retransmit, and fast recovery algorithms, RFC 2001, January 1997.
[18] S. Floyd, T. Henderson, The newreno modification to tcp's fast recovery algorithm, RFC 2582, April 1999.
[19] L.S. Brakmo, L.L. Peterson, Tcp vegas: end to end congestion avoidance on a global internet, IEEE J. Sel. Areas Commun. 13 (October 1995) 1465–1480.
[20] S. Ha, I. Rhee, L. Xu, Cubic: a new tcp-friendly high-speed tcp variant, ACM SIGOPS Oper. Syst. Rev. 42 (July 2008) 64–74.
[21] Y. Cui, L. Wang, X. Wang, H. Wang, Y. Wang, Fmtcp: a fountain code-based multipath transmission control protocol, IEEE/ACM Trans. Networking (TON) 23 (Apr. 2015) 465–478.
[22] Y. Cui, L. Wang, X. Wang, Y. Wang, F. Ren, S. Xia, End-to-end coding for tcp, IEEE Netw. 30 (March 2016) 68–73.
[23] C. Wilson, H. Ballani, T. Karagiannis, A. Rowstron, Better never than late: meeting deadlines in datacenter networks, in: Proceedings of ACM SIGCOMM, August 2011, pp. 50–61.
[24] B. Vamanan, J. Hasan, T.N. Vijaykumar, Deadline-aware datacenter tcp (d^2tcp), Proc. ACM SIGCOMM 42 (October 2012) 115–126.
[25] A. Munir, I.A. Qazi, Z.A. Uzmi, A. Mushtaq, S.N. Ismail, M.S. Iqbal, B. Khan, Minimizing flow completion times in data centers, in: Proc. of IEEE INFOCOM, April 2013, pp. 2157–2165.
[26] C.-Y. Hong, M. Caesar, P.B. Godfrey, Finishing flows quickly with preemptive scheduling, Proc. ACM SIGCOMM 42 (October 2012) 127–138.
[27] M. Alizadeh, S. Yang, M. Sharif, S. Kattin, N. McKeown, B. Prabhakar, S. Shenker, Pfabric: minimal near-optimal datacenter transport, Proc. of ACM SIGCOMM 43 (October 2013) 435–446.
[28] A. Munir, G. Baig, S.M. Irteza, I.A. Qazi, A.X. Liu, F.R. Dogar, Friends, not foes synthesizing existing transport strategies for data center networks, in: Proceedings of ACM SIGCOMM, October 2014, pp. 491–502.
[29] R. Mittal, V.T. Lam, N. Dukkipati, E. Blem, H. Wassel, M. Ghobadi, A. Vahdat, Y. Wang, D. Wetherall, D. Zats, Timely: rtt-based congestion control for the datacenter, in: Proceedings of ACM SIGCOMM, October 2015, pp. 537–550.
[30] V. Tsaoussidis, C. Zhang, Tcp-real: receiver-oriented congestion control, Comput. Netw. 40 (November 2002) 477–497.
[31] S. Liang, D. Cheriton, Tcp-rtm: using tcp for real time applications, in: Proceedings of ICNP, November, 2002.
[32] V. Jeyakumar, M. Alizadeh, D.Mazieres, B.Prabhakar, C. Kim, A. Greenberg, Eyeq: practical network performance isolation at the edge, in: Proceedings of USENIX conf. Networked Systems Design and Implementation (NSDI), April 2013, pp. 297–312.
[33] W. Bai, K. Chen, H. Wu, W. Lan, Y. Zhao, Pac: taming tcp incast congestion using proactive ack control, in: Proceedings of the 2014 IEEE 22nd International Conference on Network Protocols, October 2014, pp. 385–396.
[34] K. Ramakrishnan, S. Floyd, D. Black, The addition of explicit congestion notification (ecn) to ip, RFC 3168, September 2001.

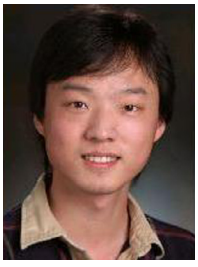


Lei Xu received his bachelor degree in computer science from Beijing Institute of Technology, China in 2006. He is working toward his Ph.D. degree supervised by Prof. Ke Xu in the Department of Computer Science & Technology at Tsinghua University. His research interests include datacenter networking and router security.

Ke Xu received his Ph.D. from the Department of Computer Science & Technology of Tsinghua University, Beijing, China, where he serves as a full professor. He has published more than 100 technical papers and holds 20 patents in the research areas of next-generation Internet, P2P systems, Internet of Things (IoT), network virtualization and optimization. He is a member of ACM and has guest-edited several special issues in IEEE and Springer Journals.

Yong Jiang received his Ph.D. from the Department of Computer Science & Technology of Tsinghua University, Beijing, China. He is currently a professor at the Graduate School at Shenzhen, Tsinghua University. He has published more than 30 technical papers on IEEE Transactions on Information Theory, Discrete Mathematics and IEICE Transactions on Communications, etc. His primary research interests include Internet architecture, Internet application, mobile Internet and the next-generation Internet.

Fengyuan Ren is a professor of the Department of Computer Science & Technology at Tsinghua University, Beijing, China. He received his B.A, M.Sc. in Automatic Control and Ph.D. in Computer Science from Northwestern Polytechnic University, Xi'an, China, in 1993, 1996 and 1999, respectively. He worked at the Department of Electronic Engineering from 2000 to 2001 and moved to the Department of Electronic Engineering as a post-doctoral researcher in Jan. 2002 both at Tsinghua University. His research interests include network traffic management and control, control in/over computer networks, wireless networks and wireless sensor networks. He (co)-authored more than 80 international journal and conference papers. He is a member of IEEE and serves as a technical program committee member and local arrangement chair for various IEEE and ACM international conferences.



Haiyang Wang received his Ph.D. from Simon Fraser University, Burnaby, BC, Canada. He is currently an assistant professor in the Department of Computer Science at the University of Minnesota Duluth, MN, USA. His research interests include cloud computing, big data, socialized content sharing, multimedia communications, peer-to-peer networks, and distributed computing.