TSINGHUA SCIENCE AND TECHNOLOGY
ISSN 1007-0214 03/11 pp491-499
Volume 21, Number 5, October 2016

SED: An SDN-Based Explicit-Deadline-Aware TCP for Cloud Data Center Networks

Yifei Lu*

Abstract: Cloud data centers now provide a plethora of rich online applications such as web search, social networking, and cloud computing. A key challenge for such applications, however, is to meet soft real-time constraints. Due to the deadline-agnostic congestion control in Transmission Control Protocol (TCP), many deadline-sensitive flows cannot finish transmission before their deadlines. In this paper, we propose an SDN-based Explicit-Deadline-aware TCP (SED) for cloud Data Center Networks (DCN). SED assigns a base rate for non-deadline flows first and gives spare bandwidth to the deadline flows as much as possible. Subsequently, a Retransmission-enhanced SED (RSED) is introduced to solve the packet-loss timeout problem. Through our experiments, we show that SED can make flows meet deadlines effectively, and that it significantly outperforms previous protocols in the cloud data center environment.

Key words: data center networks; SDN; TCP; congestion; deadline-aware

1 Introduction

In recent years, the global data center business has expanded rapidly. Various data-center-hosted services, including online services such as web search, social networks, and offline applications such as data mining based on Hadoop, have become pervasive. A user request, like a web search or Hadoop MapReduce, may cause hundreds of flows to be produced in Data Center Networks (DCN). For better interactivity, these flows are allocated diverse communication deadlines ranging from 10 ms to 100 ms[1].

In these communication processes, if some flows miss their deadlines, the data they carry is not accepted by intermediate nodes (e.g., aggregators in Hadoop) resulting in bad response quality and poor network performance. Ultimately, operator revenue is affected. For instance, Amazon sales decline 1% for every 100 ms increase in service latency[2]. However, legacy TCP, which makes up more than 95% of data center traffic[1,3], cannot provide efficient transmission services.

There are several reasons for this performance degradation. First, cloud data center services often follow a Partition/Aggregate traffic pattern, which allows the participation of, typically, thousands of servers to achieve high performance. This causes traffic bursts at aggregators. Second, Top-of-the-Rack (ToR) switches, where the server is connected, are shallow-buffered, normally having only a 3–4 MB shared packet-buffer memory. Sometimes this shallow buffer size is not enough to handle such traffic bursts, resulting in buffer overflows, which are called "TCP incast congestion". A typical scenario is shown in Fig. 1. Third, the retransmission timeout to detect incast congestion (i.e., packet losses) is too long, because TCP is designed for wide-area networks. For example, the minimum RTO of TCP is generally set to 200–300 ms, but actual Round-Trip Times (RTTs) are only hundreds of microseconds in DCN. The last reason is that legacy TCP is deadline-agnostic, resulting in failing

• Yifei Lu is with School of Computer Science and Engineering, Nanjing University of Science and Technology, Nanjing 210094, China. He is also with the Key Laboratory of Computer Network and Information Integration (Southeast University), Ministry of Education, Nanjing 210096, China. E-mail: luyifei@njust.edu.cn.

∗ To whom correspondence should be addressed.

Manuscript received: 2016-07-22; revised: 2016-08-04; accepted: 2016-09-06

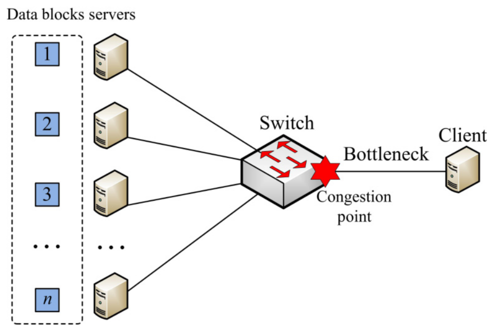


Fig. 1 Typical TCP incast scenario.

to complete transmissions in time. In summary, legacy TCP suffers from incast congestion, low goodput, and long completion times.

Currently prevailing transport protocols like TCP, DCTCP[1], RCP[4], and ICTCP[5] are deadline-agnostic. They strive to allocate bandwidth equally among flows to approximate fair sharing. The lack of awareness of flow deadlines causes a large number of flows to miss their deadlines, and the underlying reason is the tendency to treat flows equally to achieve fairness when congestion occurs. As a result, a new objective of meeting flow deadlines has inspired researchers to reinvestigate the design of TCP in DCN. Some recent works, such as D^3[2] and D^2TCP[6], introduce deadline-awareness in the TCP design. They make an effort to allocate differentiated bandwidth based on flow size and deadline. This leads to allow flows with deadlines to be sent at higher rates, so that they can complete sooner and meet their deadlines.

Software-Defined Networking (SDN)[7] is a revolutionary network architecture that separates network control functions from the underlying equipments and deploys them centrally on the controller, with OpenFlow as the standard interface. The unique characteristics of SDN make it an appropriate choice for DCN, in particular, for network management.

Using an important characteristic of SDN, which enables applications to be aware of network traffic and congestion, we propose an SDN-based Explicit-Deadline-aware TCP, called SED, for DCN. SED assigns a base rate for non-deadline flows first, to avoid "starving" non-deadline flows, and then allocates bandwidth to deadline flows, as many as possible. If the switch has spare capacity after these steps, it distributes the spare capacity fairly among non-deadline flows.

The paper is organized as follows: In Section 2, we introduce related works. In Section 3, we propose a system model to address the TCP incast problem, and describe the details of SED. A Retransmission-enhanced SED (RSED) is addressed in Section 4. In Section 5, we describe our experimental methodology and present our results. We conclude in Section 6.

2 Related Work

Many approaches to TCP congestion control have been proposed to date. In this section, we summarize the most relevant works.

Traditional Additive Increase Multiplication Decrease (AIMD) TCP achieves remarkable success in the Internet, due to the simplicity and reliability of using packet drop as congestion feedback. However, TCP reacts to the presence of congestion, rather than to its congestion level. This feature causes substantial underutilization of network bandwidth over high-speed long-distance networks. But it is important to recognize that the communication environment of DCN is significantly different from that of Wide Area Network (WAN) in terms of high bandwidth and low latency.

In the context of DCN, both DCTCP[1] and D^2TCP[6] are proposed to maintain short queue length through the sender-side back-off mechanism, to meet requirements of delay-sensitive applications. DCTCP aims to ensure low latency for short flows and good utilization for long flows by reducing switch buffer occupation, while minimizing buffer oscillation. In DCTCP, Explicit Congestion with thresholds is used for congestion notification, while both TCP sender and receiver are modified for a novel fine-grained congestion window adjustment. Reduced switch-buffer occupation can effectively mitigate potential overflow caused by TCP incast. D^2TCP builds on DCTCP and adds deadline awareness to it. It changes the congestion window update function to incorporate deadline information when congestion is detected: far-deadline flows back-off more, and near-deadline flows back-off less. In the far-deadline phase, D^2TCP backs-off more than DCTCP, and in the stable phase, D^2TCP operates very similarly to DCTCP, and gives up bandwidth if new flows join the network. However, it still cannot satisfy all of the deadline requirements. One reason is that D^2TCP uses deadline information for its back-off in near-deadline phase, when it is already too late to react to stringent deadlines.

Compared with D^2TCP, where reactive congestion control is distributed among senders, in D^3 and PDQ[8], switches become the critical controllers that proactively allocate sending rates to flows.

D^3 uses explicit rate control to apportion bandwidth according to flow deadlines. Given a flow's size and deadline, source hosts request desired rates to switches. The switches assign and reserve allocated rates for the flows. Preemptive Distributed Quick (PDQ) is a flow-scheduling algorithm designed to complete flows quickly and meet flow deadlines. PDQ emulates a Shortest Job First (SJF) algorithm to give a higher priority to the short flows. PDQ provides a distributed algorithm by allowing each switch to propagate flow information to others via explicit feedback in packet headers.

Unlike existing approaches that are either host-based approaches or network-based, DIATCP[9] is proposed under the Partition/Aggregate traffic pattern, where the aggregator is aware of the bottleneck link capacity as well as the traffic on the link. DIATCP controls the peers' sending rate directly to avoid incast congestion and to meet cloud applications' deadlines.

SDN has recently been proposed to build a "clean slate" network architecture. In such an architecture, we expect that hardware and compatibility would no longer be design constraints. In addition, with a global network view provided by SDN, control decisions can be made by a centralized controller with more accuracy. Facilitated by SDN, centralized network protocols can be designed and implemented to optimize the performance of applications with deadline requirements with finer granularity.

3 SED Algorithm

3.1 System model

In DCN, each intermediate switch or router maintains a virtual input queues at each input port and an output queue at each output port; these queues share the switch memory[10]. In this paper, the network consists of n nodes (sender), 1 node (receiver), and 1 bottleneck switch, as shown in Fig. 1.

We consider there are N flows, sharing a link of capacity C and a single switch, and we denote the congestion windows of flow i as $W_i(t)$ at time t. Then the queue size at time t, is given by

$$Q(t) = \sum_{i \in N} W_i(t) - C \times \text{RTT}_{\text{avg}} \tag{1}$$

where RTT_{avg} is the average RTT of all N flows.

3.2 SED overview

We categorize the flows in DCN into non-deadline flows that have no specific deadlines for flow completion, and deadline flows that are supposed to be completed by a specific deadline. Like D^2TCP and D^3, we assume that applications expose their size and deadline information when initiating a deadline flow, and it is reasonable that applications in DCN can be managed by network operators.

In the context of non-congestion network, TCP follows the classical AIMD mechanism. However, the deadline awareness of SED will take effect only until congestion happens. The basic rationale of our SED is to assign a base rate for non-deadline flows first at the switch, which can avoid starving the non-deadline flows, and then give bandwidth to as many of the deadline flows as possible. If the switch has spare capacity after the above steps, it distributes the spare capacity fairly among all non-deadline flows. The deadline awareness in SED is employed by the window allocation algorithm, which we will explain in Section 3.6 in detail. By doing these, we control the total amount of traffic, in order not to overflow the bottleneck link.

The overall procedure of SED contains two parts: queue congestion management on the switch side, and congestion control on the SDN controller side. A detailed description is shown in Fig. 2.

3.3 Queue congestion management

Packets that arrive at switches are served in First-In-First-Out (FIFO) order. We consider that the network enters congestion state (CNG) when predefined threshold, K, satifies $K \leqslant Q(t) \leqslant Q_{\max}$; otherwise, the network is in the normal state (NOM). When a switch enters the CNG state, a congestion trigger message is sent to the controller via the OpenFlow channel. In the same way, a congestion recovery message is delivered to the controller if the switch state returns to NOM. We show these state changes in Fig. 3.

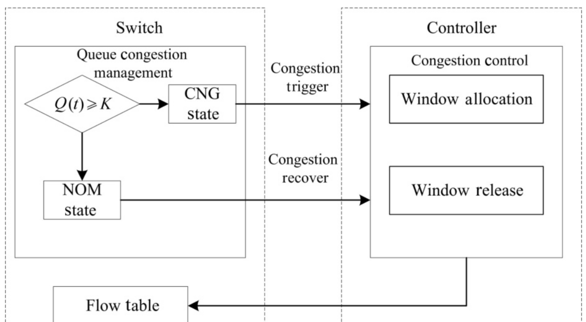


Fig. 2 The overall procedure of SED.

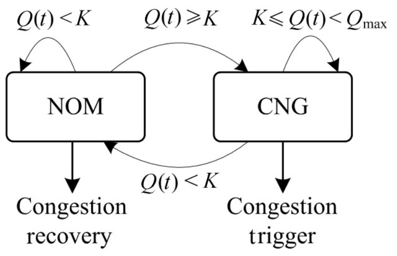


Fig. 3 State changes of SED in queue congestion management.

3.4 Congestion control at SDN controller

In an SDN controller, when receiving a congestion trigger message, we use a window allocation algorithm to meet the flow deadlines and push new flow-table entries to the switch. We utilize the receive window field in the TCP ACK header to allocate a specific window size to each sender. On the other hand, upon receiving a congestion recover message, the previous flow-table entries are deleted. In addition, when a new TCP connection is created or terminated, the window allocation algorithm will be recalled to assign new windows to each flow in the context of the congestion state. The basic congestion control mechanism is described in Algorithm 1.

3.5 Global Information Flow (GIF) table

In order to communicate between client and server, TCP uses a three-way handshake to establish a connection, and a four-way handshake for connection termination. In the establishing connection, TCP options carried in the SYN and SYN-ACK packets are used to negotiate optional functionality.

As shown in Fig. 4, a switch sends an SYN packet to a controller via a Packet_In message, when finding no matching entry in the flow table. When receiving this Packet_In message, the controller generates a routing table and pushes it to the switch. In the same way, a receiver will return an SYN-ACK packet when receiving an SYN packet. This SYN-ACK packet follows the same procedure we discussed above. In these processes, the controller records the information of the flow to form a GIF table. Figure 5 shows the detail of the GIF table.

In a GIF table, we record the time (Time) when this flow is established, and the deadline (Deadline) and flow size (Flow_size), which we can gain from applications. Subsequently, we can calculate the remaining time (RTime) until the deadline and remaining flow size (RSize) periodically, according to the OpenFlow protocol. The priority order of the GIF is sorted following EDF (Earliest Deadline First), which is known to minimize the number of late tasks, to minimize the number of missed deadline flows.

The TCP connection termination procedure is shown in Fig. 6. When the controller receives an FIN packet, it releases the resources, including deleting GIF entries and routing tables, with respect to this flow.

3.6 Windows allocation

The sender sending rate should match the link capacity from the switch to the receiver to avoid TCP incast

Algorithm 1 Congestion Control Mechanism

```
1:  if receiving congestion trigger message then
2:      state = CNG
3:      call window_allocation()
4:      push new flow table entries
5:  end if

6:  if receiving congestion recover message then
7:      state = NOR
8:      call window_release()
9:      delete flow table entries
10: end if

11: if establish (or Delete) a TCP connection then
12:     update GIF table
13:     if state = CNG then
14:         call window_allocation()
15:         push new flow table entries
16:     end if
17: end if
```

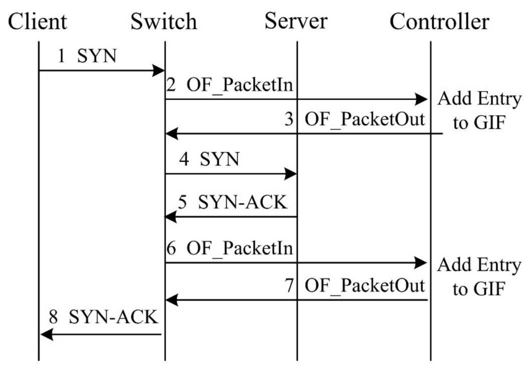
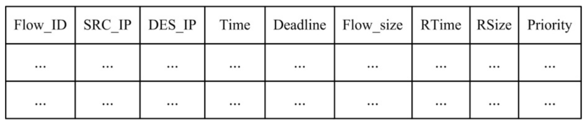


Fig. 4 GIF table generation with TCP three-way handshake.



Flow_ID	SRC_IP	DES_IP	Time	Deadline	Flow_size	RTime	RSize	Priority
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...

Fig. 5 Global information flow table.

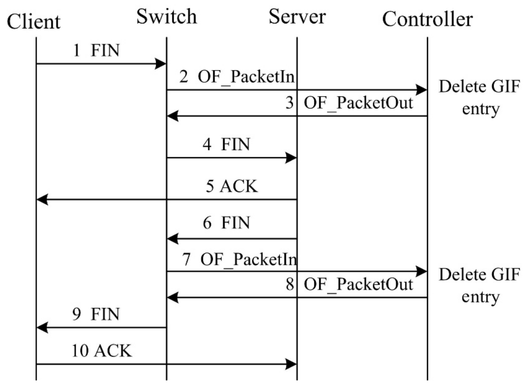


Fig. 6 GIF table deletion with TCP termination procedure.

congestion and to maintain goodput. We define the total window, referred to as Twin, as the sum of the sending window sizes of all the TCP connections passing through the switch. Therefore,

$$\text{Twin} = \sum_{i \in N} W_i(t) \tag{2}$$

From Eq. (1), we have

$$\text{Twin} = K + C \times \text{RTT}_{\text{avg}} \tag{3}$$

If a flow wants to meet its deadline, then it should follow:

$$\text{alloc window} = \frac{s}{d} \times \text{RTT}_{\text{avg}} \tag{4}$$

where s is the remaining transmit data size and d is the remaining time until the deadline.

Algorithm 2 presents the window allocation algorithm. The GIF table is ordered by giving priority to the earliest deadline flows. The flows with the earliest deadlines are allocated first. Assuming that a flow that misses its deadline is meaningless, we drop the flow if the deadline is missed (lines 2–4). Non-deadline flows are allocated to a base rate, which is usually set to 1 MSS (lines 6–8). Lines 10–12 implement the initial allocation, which corresponds to Eq. (4). Hence, the window size is allocated so that it meets the deadline of each flow. If the window requirement is larger than Twin, we set the flow's window to zero (lines 13–17). If there are remaining windows after the initial allocation, reallocation to non-deadline flows will be performed later in a fair-share manner (lines 26–28).

4 RSED

As a TCP sender transmits approximately cwnd packets within the time of RTT, the average throughput (T_{avg}) can be given by

$$T_{\text{avg}} = \frac{\text{cwnd} \times \text{MSS}}{\text{RTT}_{\text{avg}}} \tag{5}$$

Algorithm 2 Window Allocation Algorithm

```
Require:
    flow.rtime: remaining time until deadline
    flow.size: remaining data size
    flow.win: allocated window
    total_alloc = 0, req_alloc = 0;
1:  for all each flow in GIF do
2:      if flow expires then
3:          Drop this flow
4:      end if
5:      // flow.rtime = 0 for non-deadline flows
6:      if flow.rtime = 0 then
7:          flow.win = base_win //get a base window
8:          total_alloc = total_alloc + base_win
9:      else
10:         if total_alloc < Twin then
11:             req_alloc = flow.size/flow.rtime * RTT
12:             total_alloc = total_alloc + req_alloc
13:             if total_alloc > Twin then
14:                 // there is not enough windows to allocate
15:                 total_alloc = total_alloc – req_alloc
16:                 flow.win = zero
17:             else
18:                 flow.win = req_alloc
19:             end if
20:         else
21:             // there is not enough windows to allocate
22:             flow.win = zero
23:         end if
24:     end if
25: end for
26: if total_alloc < Twin then
27:     allocate the remaining window to non-deadline flows in a
        fair-share manner
28: end if
```

where the default MSS is 1460 byte.

Hence, we know that the range of cwnd can be given by $1 \leqslant \text{cwnd} \leqslant \frac{T_{\text{avg}} \times \text{RTT}_{\text{avg}}}{\text{MSS}}$. For a typical DCN, the bandwidth is 1 Gbps and the average RTT is about 200 μs. Then cwnd $\approx$ 16.7, so cwnd $\in [1, 17]$.

From the perspective of the switch, we can also get $\text{Twin} = \frac{30 \times \text{MSS} + 1\,\text{Gbps} \times 200\,\mu\text{s} \times 0.125}{\text{MSS}} \approx 46.7$ from Eq. (3) in the above typical DCN scenario, where $K = 30$ packets and switch queue size is 100 packets. In the extreme case when cwnd of each flow is 1, we know that the maximum number of concurrent flows can reach about 46.

However, the number of concurrent flows in typical DCN is far greater than 46. For example, Yahoo!'s M45 MapReduce cluster[11,12] reports that each job consists of an average of 153 Maps and 19 Reduces. A Google web search cluster reports that every query operates on data spanning thousands of servers, where a single query reads hundreds of megabytes on average[6,13]. With this in mind, we argue that packet loss is inevitable when the number of concurrent flows becomes large. When packet loss happens, the sending server receives triple duplicate ACKs, decreases its congestion window, and goes into fast recovery mode. On the other hand, the cwnd of each flow is no greater than 17, and in many cases, cwnd = 1 when concurrent flows are large, resulting in the terrible Full window Loss Timeout (FLoss-TO) and Lack of ACKs Timeout (LAck-TO)[14,15]. This phenomenon leads to TCP RTO timeout and causes a significant throughput collapse.

As a result, in this section, we propose RSED to retransmit lost packets quickly. The basic idea of RSED is that when packet loss happens in a switch, a packet-loss message to the controller will be triggered via an OpenFlow channel, resulting in triple duplicate ACKs being generated by the controller.

The queue congestion management can be extended as shown in Fig. 7. Packets are dropped when the switch queue size is greater than the switch buffer. Moreover, a packet-loss message, which is encapsulated in an OpenFlow Packet_In message, is triggered and transmitted to the controller. After obtaining the dropped packet extracted from this Packet_In message, the controller sends triple duplicate ACKs to the source of the dropped packet. Ultimately, the sender can retransmit this packet without a TCP RTO timeout.

5 Experimental Results

5.1 Setup of experiments

In this section, we describe a series of experiments in the Mininet v2.2.1[16], using Floodlight[17] as the controller and Open vSwitch v2.3.0 (OVS)[18] as the OpenFlow switch. The experiments are simulated on a server where the hardware profile includes 2.4 GHz Intel CPUs with 8 cores, 16 GB RAM, and a 1 TB hard disk, and the operating system is Ubuntu 14.04.2

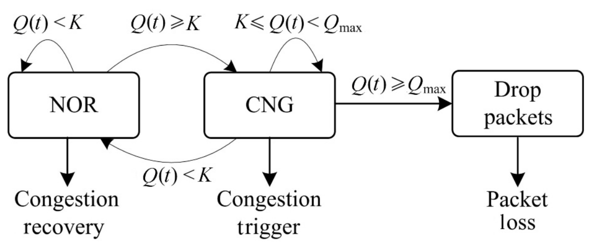


Fig. 7 State change of RSED at switch.

(kernel 3.16.0-30-generic).

Our SDN controller is implemented on top of the Floodlight platform that is deployed in a laptop with a 1.9 GHz Intel I5 Core, with 4 GB RAM, and a 500 TB hard disk. The operating system is also Ubuntu 14.04.2. For DCTCP implementation, we use public code from Ref. [19] and add ECN capability to SYN packets[20]. Meanwhile, we use TCP New Reno[21] (named TCP for short in the later experiments) as our congestion control algorithm, and disable the delayed ACK.

For the key parameters of DCTCP, we set g, the weighted averaging factor, to 1/16, and K, the buffer occupancy threshold for marking CE-bits, to 20. For D^{2}TCP, we set d, the deadline imminence factor, to be between 0.5 and 2.0, following Ref. [6]. The minimum RTO for all TCP protocols is 30 ms. We set experiment parameters as shown in Table 1.

5.2 Results

(1) ***Small-scale experiments*** In this experiment we have six senders transmitting flows to a receiver; one has no deadline, and the others have deadlines. We choose flow sizes and deadlines to illustrate the impact of a deadline-aware protocol. We set the five deadline flow sizes to 8 MB, 12 MB, 30 MB, 50 MB, and 64 MB, with respective deadlines of 300 ms, 800 ms, 1 s, 3 s, and 5 s. The flow without a deadline has infinite data to send. This topology is shown in Fig. 8.

Table 1 The experiment parameters.

Parameter	Value
Capacity of links	1 Gbps
Buffer size of each switch port	150 KB
Minimum RTO of all TCPs	30 ms
Packet size	1500 KB
MSS	1460 KB
RTT	200 μs

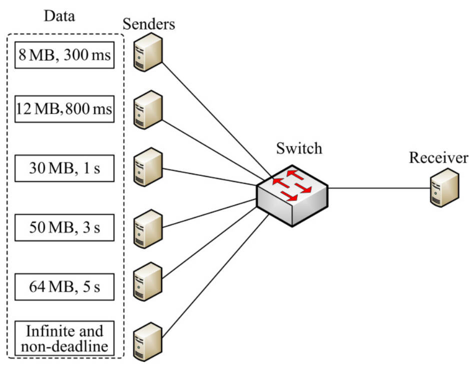


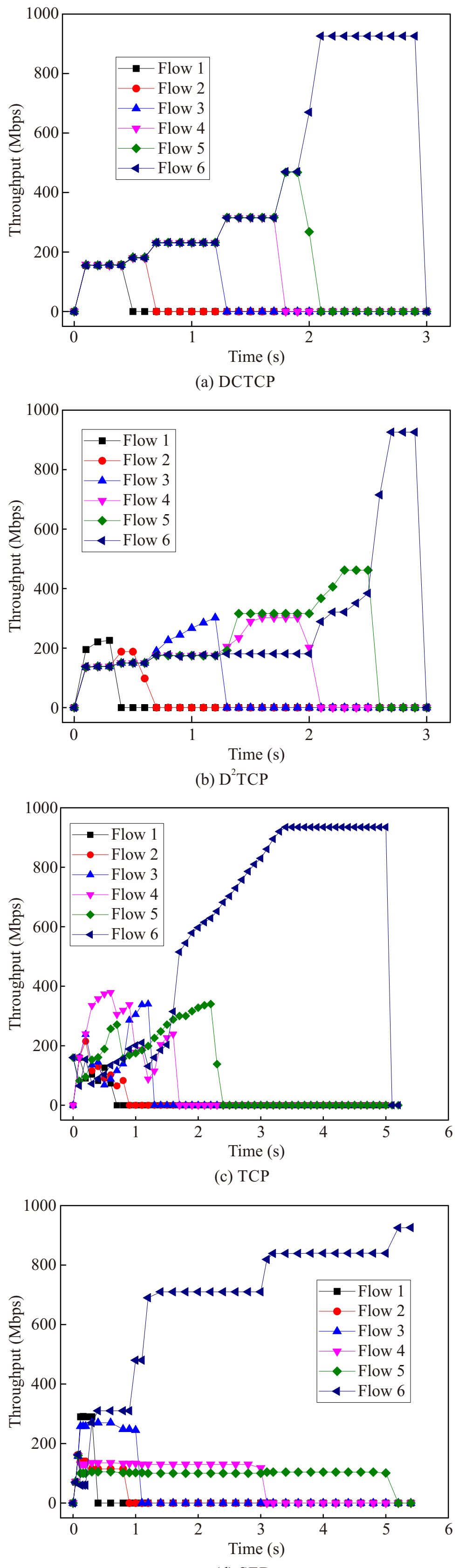
Fig. 8 Small-scale experiments topology.

In Fig. 9 we show the throughput achieved by the six flows over time, for TCP, DCTCP, D^{2}TCP, and SED. The difference between the various TCPs is most noteworthy in the 0–3 s range. Figure 9a shows that DCTCP grants all flows equal bandwidth, and consequently flow 1 and flow 3 miss their deadlines. Figure 9b shows that D^{2}TCP's deadline-aware congestion avoidance allows the near-deadline flows to take a larger share of the available bandwidth, and the far-deadline flows commensurately relinquish bandwidth. However, it also misses the deadline of flow 3. DCTCP and D^{2}TCP provide low latency with very low buffer occupancies, while still achieving high throughput. Hence, the completion time of all flows is shorter than TCP and SED. Flows 1, 2, and 3 with TCP miss their deadlines, as shown in Fig. 9c. TCP is the worst of the four transmission protocols. SED meets all the deadlines of the six flows, although it takes the longest transmission time. It is because SED allocates transmission rate according to dividing remaining time by the remaining transmit data size, so transmission will last until the deadline.

(2) ***Large-scale experiments*** We ran a set of five deadline-sensitive applications on the network, equally dividing the total number of hosts among the applications. Each application consists of one receiver and *n* senders, which have the same settings for size and deadlines. This experiment topology is shown in Fig. 10. We varied *n*, the number of senders per application, to explore varying degrees of fan-in-bursts.

In this experiment, we set the five applications' flow sizes to 20 KB, 60 KB, 100 KB, 140 KB, and 200 KB, and deadlines to 200 ms, 300 ms, 350 ms, 400 ms, and 450 ms, respectively. All TCP, DCTCP, and D^{2}TCP parameters match those in Section 5.1.

Figure 11 shows the goodput of SED and RSED with TCP, DCTCP, and D^{2}TCP as we vary the number of concurrent flows up to 100. As shown in the figure, the goodput of TCP collapses when the number of senders is larger than about 5. This phenomenon of goodput collapsing in DCTCP and D^{2}TCP happens when concurrent numbers reach above 25 and 30 respectively. SED performs well as the number of senders increases to 40. At that time, the link utilization is about 90%. Subsequently, as the number of senders continues to expand, the goodput of SED declines on account of TCP RTO timeouts caused by packet loss and missed deadlines of TCP flows. However, RSED significantly outperforms SED, TCP, DCTCP, and



(a) DCTCP

(b) D^{2}TCP

(c) TCP

(d) SED

Fig. 9 Throughput for TCP, DCTCP, D2TCP, and SED.

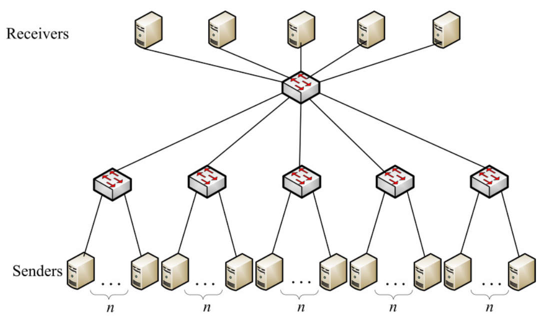


Fig. 10 Large-scale experiments topology.

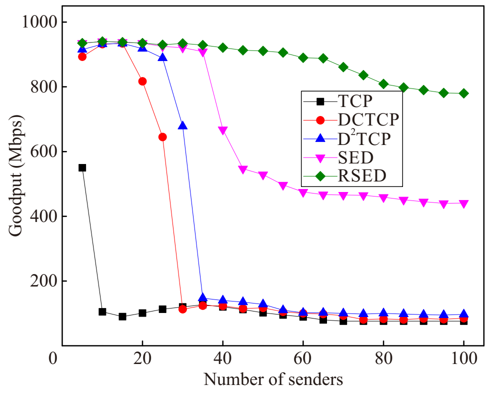


Fig. 11 Goodput for TCP, DCTCP, D^{2}TCP, SED, and RSED with concurrent senders.

D^{2}TCP when the concurrent flows are greater than 40. This is because that RSED exploits fast retransmission of lost packets to avoid TCP RTO timeout, which will decrease the goodput of TCP.

In our experiment, SED easily handled 40 concurrent flows without any performance degradation. However, RSED can significantly improve the performance of TCP, DCTCP, and D^{2}TCP over TCP incast and deadline scenarios.

Figure 12 shows the fraction of flows that miss the deadlines with increasing congestion levels. In this figure, the Y axis shows the fraction of missed deadlines for TCP, DCTCP, D^{2}TCP, SED, and RSED as we vary the degree of burstiness on the X axis by increasing the number of concurrent flows from 5 to 100.

When the number of senders is small (e.g., 10 or fewer), all variants meet the deadlines well, but the missed deadlines of TCP and DCTCP increase rapidly as the number of flows increases. D^{2}TCP performs much better than TCP and DCTCP as it gives more bandwidth to near-deadline flows, but still misses about 30% of the deadlines when the number of senders is large (e.g., 50). On the other hand, SED does not miss any deadlines even in highly congested situations. We

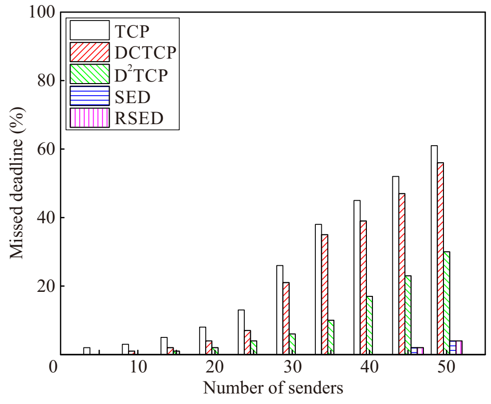


Fig. 12 Fraction of flows that miss deadlines.

note that RSED also shows similar results; it missed only 1 and 3 deadlines when the number of flows was 45 and 50, respectively. This implies that most deadlines can be met.

Figure 13 shows how incast congestion affects performance, and we measure the fraction of flows that suffer at least one timeout. It is observed that more than 20% of flows that employ TCP or DCTCP experience network congestion when the number of senders is greater than 20. D^{2}TCP shows better performance with regard to congestion avoidance, but the fraction of timeout flows increases up to around 50% as the number of senders increases. Through comparing Fig. 12 with Fig. 13, we can see that incast congestion directly affects the missed deadlines as flow deadlines range from 20 ms to 60 ms while minimum RTO is 30 ms in our experiment. Due to fact that the basic idea of SED is to avoid congestion by controlling the receive window of each flow, SED and RSED control the total sending window size to the extent of the bottleneck link capacity and as a result, suffer some timeouts.

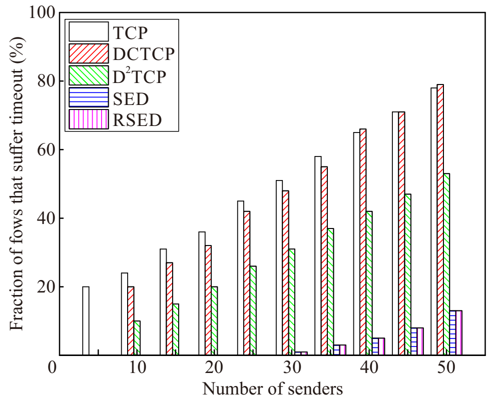


Fig. 13 Fraction of flows that suffer at least one timeout.

6 Conclusion

In this paper, we propose SED, a new SDN-based explicit-deadline-aware TCP, designed for cloud data center networks. Unlike existing approaches that are either host-based or network-based, we develop and design an SDN-based solution. Our insight is that in the SDN environment, the SDN controller is aware of the bottleneck link capacity as well as the traffic on the link. Therefore, SED controls the peers' sending rate directly to avoid TCP incast congestion and to meet the application deadline. Furthermore, a retransmission-enhanced SED, which is termed RSED, is proposed to deal with TCP RTO timeout problems caused by packet loss. We evaluate SED via extensive simulations. Our results confirm that SED can make flows meet deadlines effectively without starving the non-deadline flows.

As future work, we plan to design an optimized tuning algorithm for Twin based on mathematical analysis and to calculate deadline flow precedence.

Acknowledgment

This research was partially supported by the National Natural Science Foundation of China (Nos. 61370209 and 61402230). The author thanks Dr. Ling Tang for the valuable comments and discussions.

References

[1] M. Alizadeh, A. Greenberg, D. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, Data center TCP (DCTCP), *ACM SIGCOMM Computer Communication Review,* vol. 40, no. 4, pp. 63–74, 2010.

[2] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowtron, Better never than late: Meeting deadlines in datacenter networks, *ACM SIGCOMM Computer Communication Review,* vol. 41, no. 4, pp. 50–61, 2011.

[3] T. Benson, A. Anand, A. Akella, and M. Zhang, Understanding data center traffic characteristics, *ACM SIGCOMM Computer Communication Review,* vol. 40, no. 1, pp. 92–99, 2010.

[4] A. 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, 2006.

[5] H. Wu, Z. Feng, C. Guo, and Y. Zhang, ICTCP: Incast congestion control for TCP in data center networks, in *Proc. the ACM CoNEXT 2010*, Philadelphia, PA, USA, 2010, p. 13.

[6] B. Vamanan, J. Hasan, and T. N. Vijaykumar, Deadline-aware datacenter tcp, *ACM SIGCOMM Computer Communication Review,* vol. 42, no. 4, pp. 115–126, 2012.

[7] N. Mckeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, OpenFlow: Enabling innovation in campus networks, *ACM SIGCOMM Computer Communication Review,* vol. 38, no. 2, pp. 69–74, 2008.

[8] C. Hong, M. Caesar, and P. Godfrey, Finishing flows quickly with preemptive scheduling, in *Proc. the ACM SIGCOMM 2012 Conf. on Applications, Technologies, Architectures, and Protocols for Computer Communication*, Helsinki, Finland, 2012, pp. 127–138.

[9] J. Hwanga, J. Yoob, and N. Choi, Deadline and incast aware TCP for cloud data center networks, *Computer Networks,* vol. 68, no. 5, pp. 20–34, 2008.

[10] N. McKeown, A fast switched backplane for a gigabit switched router, http://www.cs.cmu.edu/∼srini/15-744/papers/McK97.html, 1997.

[11] Yahoo! m45 supercomputing project, http://research.yahoo.com/node/1884, 2007.

[12] S. Kavulya, J. Tan, R. Gandhi, and P. Narasimhan, An analysis of traces from a production mapreduce cluster, in *Proc. the 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, Washington DC, USA, 2010, pp. 94–103.

[13] L. Barroso, J. Dean, and U. Holzle, Web search for a planet: The google cluster architecture, *IEEE Micro,* vol. 23, no. 2, pp. 22–28, 2003.

[14] J. Zhang, F. Ren, and C. Lin, Modeling and understanding TCP incast in data center networks, in *Proc. IEEE INFOCOM 2011*, Shanghai, China, 2011, pp. 1377–1385.

[15] J. Zhang, F. Ren, L. Tang, and C. Lin, Taming tcp incast throughput collapse in data center networks, in *Proc. 21st IEEE International Conference on Network Protocols (ICNP)*, Goettingen, Germany, 2013, pp. 1–10.

[16] Mininet, http://mininet.org/, 2016.

[17] Floodlight, http://www.projectfloodlight.org/floodlight/, 2016.

[18] Open vSwitch, http://openvswitch.org/, 2016.

[19] DCTCP Patch, http://simula.stanford.edu/∼alizade/Site/DCTCP.html, 2016.

[20] A. Kuzmanovic, A. Mondal, S. Floyd, and K. Ramakrishnan, Adding Explicit Congestion Notification (ECN) capability to TCP's SYN/ACK packets, https://tools.ietf.org/html/rfc5562, 2016.

[21] F. Sally and H. Tom, The NewReno modification to TCP's fast recovery algorithm, https://tools.ietf.org/html/rfc2582, 2009.

Yifei Lu received the PhD degree from Southeast University in 2010. He is now a lecturer in Nanjing University of Science and Technology. His main research interests include software-defined networking and data center network.