Catch the Whole Lot in an Action: Rapid Precise Packet Loss Notification in Data Center

Peng Cheng, Fengyuan Ren, Ran Shu, and Chuang Lin, *Tsinghua University*

https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/cheng

This paper is included in the Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI '14).

April 2–4, 2014 • Seattle, WA, USA

ISBN 978-1-931971-09-6

Open access to the Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI '14) is sponsored by USENIX

Catch the Whole Lot in an Action: Rapid Precise Packet Loss Notification in Data Centers

Peng Cheng, Fengyuan Ren, Ran Shu, Chuang Lin
Dept. of Computer Science and Technology, Tsinghua University, Beijing, 100084, China
Email: {chengpeng5555, renfy, shuran, clin}@csnet1.cs.tsinghua.edu.cn

Abstract

An increasing number of TCP performance issues including TCP Incast, TCP Outcast, and long query completion times are common in large-scale data centers. We demonstrate that the root cause of these problems is that existing techniques are unable to maintain self-clocking or to achieve accurate and rapid packet loss notification. We present cutting payload (CP), a mechanism that simply drops a packet's payload at an overloaded switch, and a SACK-like precise ACK (PACK) mechanism to accurately inform senders about lost packets. Experiments demonstrate that CP successfully addresses the root cause of TCP performance issues. Furthermore, CP works well with other TCP variants used in data center networks.

1 Introduction

Modern large-scale data centers, which enable cloud computing and host online services with intensive server-side computing and storage, are significantly different from traditional data centers because of shorter round trip time (RTT), higher bandwidth, highly variable flow characteristics [4, 18] and very low latency requirements [4, 28]. Because of these differences, TCP has been found to have various performance issues in large-scale data center networks (DCNs).

In recent years, TCP Incast [22], TCP Outcast [24], the TCP out-of-order problem [32], and long-query completion times [4] have been found to significantly affect TCP performance in DCNs. Through experiments, we found that throughput cliff (described in § 2.1) and TCP unfairness in multiple-bottleneck scenarios [20] may also exist in DCNs. Collectively, these problems are referred to as "TCP problems" in this paper. These problems must be resolved to achieve high throughput and low latency, which are critical requirements of many data center applications [4, 14].

Our experimental observations and comprehensive analysis attribute TCP problems to three issues. First, self-clocking stall caused by insufficient ACKs results in Incast, throughput cliff, and unfairness. Second, inaccurate packet-loss notification caused by ambiguous loss indication results in the TCP out-of-order problem. Third, slow packet loss detection leads to long query completion time.

Packet loss notification is a powerful mechanism to address these problems. It maintains self-clocking, makes unambiguous differentiation between packet loss and out-of-order packets and shortens the detection time of packet loss. We propose a simple solution called CP[1] for packet loss notification. CP drops only the packet payload instead of the entire packet during buffer overload and uses a SACK-like precise ACK (PACK) technique to accurately inform senders of lost packets. In our experiments, CP successfully demonstrates its ability to solve TCP problems.

We implement CP in NetFPGA cards. In our implementation, CP only results in a 56-ns processing delay and less than a 2% increase in resource usage. Furthermore, the additional overhead is rarely introduced because the processing is only triggered by packet loss. Because of its low overhead and limited extra resource usage, CP can be easily added to existing commercial switches.

This paper makes two main contributions:

- Our comprehensive analysis identifies three key issues with TCP in large-scale DCNs: self-clocking stall, inaccurate packet loss notification, and slow packet loss detection.
- We propose CP, which leaves the packet header intact during packet drop processing, merely cutting out the payload to rapidly inform the sender of packet loss. Experiments demonstrate that this can solve many TCP problems and has good compatibility with other variations of TCP used in DCNs.

[1] CP is the abbreviation of "cutting payload".

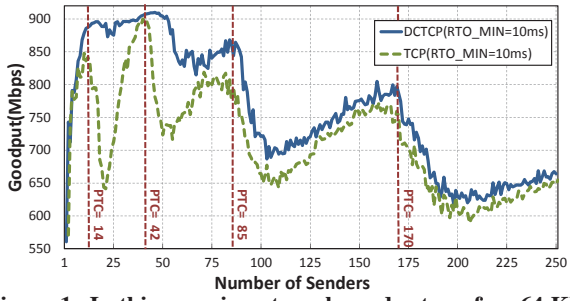


Figure 1: In this experiment, each sender transfers 64 KB data to one receiver through 128 KB bottleneck link simultaneously. The network topology and other experimental configurations are the same as MapReduce-like application scenario described in § 6.1. We observe that DCTCP avoids the throughput cliff where PTC=14 and postpones it from PTC=42 to PTC=50, but still suffers the throughput cliff with an increasing number of senders.

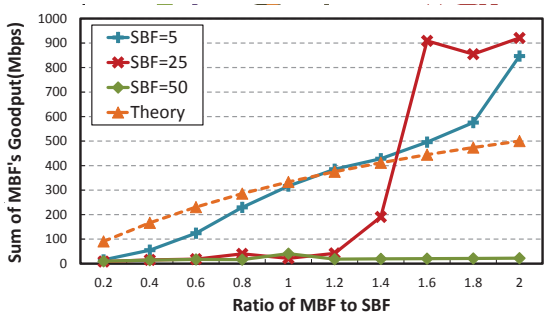


Figure 2: In this experiment, all flows use TCP with SACK, and the topology is shown in Fig. 9 (b). Other experimental configuration are the same as those described in § 6.2.

The remainder of this paper is organized as follows. In Section 2, we summarize the existing TCP problems. In Section 3, we analyze inherent reasons for these problems. In Section 4 and Section 5, we propose CP and describe its design and implementation in detail. In Section 6, we test and illustrate the performance of CP when handling TCP performance issues in DCNs. Finally, we conclude the paper in Section 7.

2 TCP in Contemporary DCNs

In this section, we summarize and discuss TCP problems in DCNs based on experimental results from a small testbed. Our testbed is made up of one aggregation switch, four top-of-rack (ToR) switches, and twelve end-hosts. More details are discussed in § 6.

2.1 Low and Volatile Throughput

In current data centers, the many-to-one communication pattern is common in applications such as MapReduce [10] and Web search applications [18]. In this pattern, data from many synchronized senders is transferred to the same receiver in parallel; thus, TCP Incast collapse naturally occurs and throughput decreases sharply. Many approaches have been proposed to address TCP Incast, such as DCTCP [4], ICTCP [29], and decreasing RTO$_{min}$ [27]. However, through experimental observation, we found that these approaches are not able to satisfy the particular demands of two major types of applications: MapReduce-like applications and Web search applications.

In MapReduce-like applications, the number of senders varies dramatically (the average is 154 with a standard deviation of 558) [19]. When the number of senders is large, the buffer associated with the bottleneck link overflows even though each sender transmits only one packet [4]. Then, because of packet loss and the one-packet transmission window, timeouts are inevitable. In order to keep high throughput and avoid TCP Incast collapse, even with DCTCP and ICTCP, the number of senders needs to stay within an upper limit. In practice, this limit fails to meet demands of large-scale applications. In addition, we observed a particular phenomenon when using a small value for RTO$_{min}$, as shown in Fig. 1. We call this a throughput cliff, reflecting that the throughput will sharply decline at the point of throughput cliff (PTC) and then slowly climb. We found that the throughput cliff is due to synchronous adjustment of the congestion window at senders and synchronous packet loss[2]. The synchronous packet loss leads to increasing timeouts and throughput decline. Thus, these well-known approaches to alleviate TCP Incast collapse cannot maintain sufficiently high throughput with a large number of senders.

In Web search applications, because of the fixed size of search result, the number of senders is not significantly large. However, the query completion time is closely related to Web search performance [28]. From our experiments, we were surprised to find that, because a high and stable throughput cannot be guaranteed by TCP or its variants, query completion time is far greater than expected and has frequent fluctuations[3].

2.2 TCP Unfairness

TCP unfairness in wide area networks and wireless networks is well known. In DCNs, TCP unfairness occurs in both single-bottleneck and multi-bottleneck scenarios.

In a single-bottleneck scenario, a port blackout [24], in which each input port accidently loses a series of packets, causes the consecutive packets loss or even entire window loss leading to TCP timeout. When there are

[2]For example, with 14 senders, if the bottleneck buffer is 128 KB and each sender increases its congestion window to 7 packets ($1.5KB \times 14 \times 6 = 126KB$), all senders will see a drop synchronously. Consequently, synchronous adjustment of congestion window and synchronous packet loss happen at PTC.

[3]The concrete results are presented in Fig. 8(b) in § 6.1. Using TCP with 10ms-RTO$_{min}$, there is a delay of twice the expected value when the number of senders is greater than 26. Using DCTCP, there is a severe fluctuation in query completion time when the number of senders is between 22 and 42.

both a large and small number of flows from two different input ports to one output port, timeouts preferentially happen on the small flows, which results in TCP Outcast. A straightforward solution called equal-length routing [24] changes the routing paths through core switches to address this problem. However, this imposes significant pressure on core switches and increases end-to-end delay. Therefore, alleviating or eliminating the damage caused by a port blackout without imposing additional costs is desirable.

In the multiple-bottleneck scenario, multiple-bottleneck flows (MBF), i.e., cross-rack flows, share two bottlenecks with two different single-bottleneck flows (SBF), i.e., in-rack flows, which have the same number of flows. Fig. 2 shows that when the packet loss ratio is very high (i.e., SBFs = 50), MBFs have a significantly lower throughput. However, when the packet loss ratio is low (i.e., SBFs = 5 or 25), the throughput of MBFs is higher than the theoretical value. Further analysis indicates that MBFs always have a higher packet loss ratio than SBFs; thus, they may lose all transmitted packets and trigger timeouts in a high packet loss ratio scenario. However, in a low packet loss ratio scenario, SBFs suffer the TCP Outcast problem, which leads to the higher throughput of MBFs. Thus, in both situations, either SBFs or MBFs suffer from unfairness.

2.3 TCP Out-of-order Problem

Several data center topologies [1, 14, 15] have been proposed to deal with bottlenecks in core switches; however, they employ many redundant links that are utilized to only 40% -50% [8, 25]. Therefore, many multipath routing techniques have been proposed, such as ECMP, Hedera [2], and Valiant Load Balancing (VLB) [14]. All these techniques are flow-based traffic splitting schemes, which are dramatically worse than an ideal packet-level one. However, packet-level schemes lead to serious throughput degradation because of out-of-order packets [11]. Therefore, tackling the out-of-order problem can significantly improve the performance of multipath routing techniques.

2.4 Long Query Completion Time

Data centers host many soft real-time online services such as retail, advertisement, and Web search[28]. Query completion time is very critical because it is directly related to the quality of these online services.

Query completion time is influenced by the amount of queuing delay and retransmission time. Long queuing delay has been addressed by some proposed schemes [4, 5, 28]. We conducted an experiment to determine if query completion time is affected by a long retransmission time. In our experiment, we ran a query with 20 flows. As shown in Fig. 3, in flow No.18, the delay

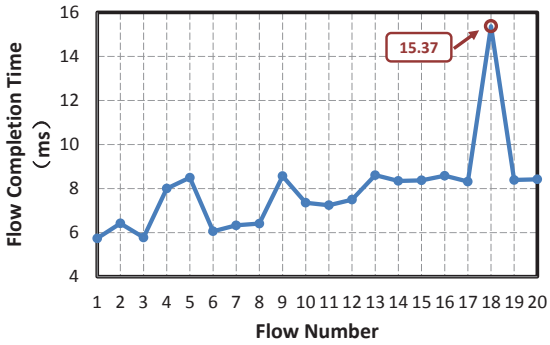


Figure 3: Each of 20 DCTCP flows transfers 50 KB to the same receiver through the 128 KB bottleneck link. We conducted experiments for 10,000 random instances, and found that in 1,231 experiments, some flow completion times are larger than 15ms. The figure shows detailed information of the completion time of each flow in one of these experiments.

is approximately two times that of other flows, thereby resulting in delay of the completion time of the entire query. After more in-depth observation, we found that flow No.18 suffers from many fast retransmissions and wastes significant time retransmitting. More than 12% of the queries suffer the same problem in our 10,000 random experimental instances. Therefore, we conclude that retransmission delay imposes a significant influence on query completion time.

3 Why TCP Does Not Work Well in DCNs

Here, we discuss three fundamental weaknesses of TCP and its variants used in DCNs.

3.1 Self-clocking Stall

The self-clocking mechanism was proposed in 1988 [17]. Essentially, the arrival of an ACK tells the sender that the network can accept another packet. Through this mechanism, TCP can maintain continuous and stable transmission and quickly fill up the pipeline. Without sufficient ACKs, however, self-clocking stops, which causes a timeout. In DCNs, the congestion window of each flow is relatively small because of the low delay-bandwidth product and large number of concurrent connections. In addition, the many-to-one traffic pattern often causes severe congestion and a high packet loss ratio. Therefore, self-clocking stall in a many-to-one traffic pattern often triggers timeouts that result in low and volatile TCP throughput. The port blackout phenomenon and high packet loss ratio of MBFs lead to the loss of the vast majority of packets in a window or even the entire window. This makes it increasingly less likely that a flow can receive a sufficient number of ACKs to maintain self-clocking. Therefore, self-clocking stall is an essential reason for low and volatile TCP throughput and TCP unfairness.

To maintain self-clocking, three enhancements have been proposed in prior work. The first mechanism is defined in RFC 3042 [7]: when a sender receives two duplicate ACKs, it sends one more packet immediately. The second mechanism is the TCP implementation in Linux kernel 2.6.3, in which a sender sends one more packet every time it receives a duplicate ACK. Unfortunately, the two mechanisms mentioned above fail when all segments in whole congestion window are lost, such as with the port blackout phenomenon. The third mechanism is tail loss probe (TLP) [12]. It transmits one packet every two round-trips when no ACK is received at the end of the transaction. This mechanism is an incomplete solution to avoid self-clocking stall because it still fails except in the tail loss case. Therefore, whether we use standard TCP, RFC 3042, TCP in Linux, or TLP, self-clocking stall remains inevitable.

This leads to the realization that we need a complementary method, apart from ACK, to maintain self-clocking for addressing TCP Incast and unfairness.

3.2 Inaccurate Packet-loss Notification

In § 2.3, we see that packet-level multipath routing techniques cause performance problems due to TCP's reaction to out-of-order packets. Here, by considering the features of DCNs, we provide further explanation. Since queuing delay causes most of the end-to-end delay in DCNs, a high-latency path is always accompanied by link congestion. Therefore, the phenomena of packet loss and out-of-order packets often co-exist in multipath DCNs. TCP deals poorly with these mixed paths due to the one-size-fits-all solution of using a fixed threshold value (three duplicate ACKs) to determine whether packet is out-of-order or lost. This often leads to spurious retransmission or sluggish congestion control [32] because it is impossible to determine the exact number of allowable out-of-order packets to set an effective threshold. TCP SACK is an effective way to avoid retransmitting received packets, however, it also cannot address this problem because it is not able to distinguish whether a packet is out-of-order or lost. Therefore, only by accurately distinguishing between lost or out-of-order packets, can TCP make correct decisions regarding packet retransmission and congestion window size.

3.3 Slow Packet loss Detection

In § 2.4, we claimed that reducing retransmission delay is important to shorten query completion time. Retransmission delay is composed of the detection time of packet loss and the retransmission time of lost packets. However, it is difficult to improve the retransmission time of lost packets because it depends on a relatively mature congestion control algorithm. Thus, we focus on reducing detection time.

There are two indicators of packet loss in TCP: timeout and three duplicate ACKs. The detection time of packet loss caused by timeout is directly related to RTT estimations and RTO_{min}. Imprecise RTT estimations [31] and improper RTO_{min} [27] lead to slow packet loss detection. Timeouts lead to other problems as well [31]; thus, avoiding timeouts is a good strategy, as discussed in § 3.1.

In the traditional Internet, packet loss is detected by the reception of three duplicate ACKs. In DCNs, due to the small congestion window, it is very unlikely that the window size is large enough to cause enough duplicate ACKs to be received by the sender in one RTT. Moreover, packet loss implies congestion where RTT becomes very long as a result of a crowded queue. Therefore, timeouts and the need for three duplicate ACKs further delays packet loss detection.

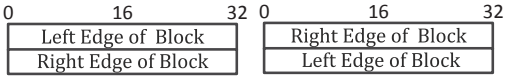
3.4 Related Work

There are three typical types of schemes to address these problems.

One is to use a TCP-like protocol, such as DCTCP. DCTCP uses ECN to adjust the congestion window. Unfortunately, when packets are dropped by the switch, the ECN information is also lost and DCTCP degenerates to standard TCP or SACK, which do not work well in DCNs. DCTCP also does not work normally in such an environment, especially with the loss of the entire congestion window. Furthermore, DCTCP does not work well in a multipath environment because of the potential for out-of-order packet. Finally, DCTCP must wait to receive three packets before retransmission when packet loss occurs, and this period can be relatively long because of potential queuing delays on switches. Both HULL [5] and D^2TCP [26], which extend DCTCP, have these same problems.

The second class of schemes is based on rate control. QCN [3], D^3 [28] and PDQ [16] use implicit or explicit rate control to avoid self-clocking stall. However, they cannot achieve accurate and rapid packet loss detection. pFabric [6] is a clean-slate design that totally solves all the above problems. Using rapid retransmission by reducing RTO_{min} to one RTT, pFabric avoids inaccurate and slow packet loss detection. However, it is hard to implement it in practice because it needs changes to both endhosts and switch hardware.

The third class of schemes uses special packets to inform the sender of a packet loss. For example, source quench [23] sends ICMP packets for every loss. The problem with such schemes is that it is very difficult for a switch to create a packet to be sent in the reverse direction when traversing the forward path. It also destroys TCP self-clocking and self-pacing.



(a) SACK (b) PACK

Figure 4: Unambiguous identities of SACK and PACK.

4 CP Design

We now discuss the design of CP.

4.1 Packet Types

CP uses four types of packets.

(1) Normal packets are sent by a sender that does not support CP. These packets are dropped by a switch when the buffer exceeds the defined threshold.

(2) CP-enabled packets are data packets with payloads that are sent by CP-capable senders. Only the payload portion of a CP-enabled packet is dropped when the buffer exceeds the defined threshold.

(3) Payload-cut packets are header-only packets without payloads. A CP-enabled packet becomes a payload-cut packet after payloads are cut off by the switch. Switches should ignore the IP length field in payload-cut packets.

(4) PACKs contain accurate packet loss information and are sent by CP-capable receivers when payload-cut packets are received.

4.2 CP Drop Processing

A switch with the payload-cutting function is called a CP switch. A CP switch has a buffer for each port to store packets awaiting forwarding. If the total size of packets in any port buffer exceeds the threshold, and a new CP-enabled packet arrives, the switch will cut off the payload of the CP-enabled packet. During this operation, first, the packet is marked as payload-cut packet. Then, the IP length is preserved for the receiver to calculate the packet size (further explanations in § 4.3), and the TCP checksum is recalculated and revised. After that, the regenerated packet without the payload queues in the buffer as a normal packet and waits for forwarding. It should be noted that a little extra buffer space is necessary to store new payload-cut packets. This will be further discussed in § 4.6.

Because of IP length field conservation, payload-cut packets may be dropped at modern switch ASICs or many middle boxes. As an alternative to achieve CP drop processing, we can preserve the first two bytes of the data payload to carry the IP length. In consideration of the similarity between these two methods, we will just discuss the first method.

4.3 Packet-loss Feedback: PACK

For backwards compatibility, we define a PACK option similar to the SACK option. As shown in Fig. 4(a), the SACK option uses the left and right edges of the block to indicate a SACKed block. The left edge represents the first sequence number of the block, and the right edge corresponds to the sequence number immediately following the last seen sequence number [21].

There are two key considerations to accomplish designing PACK. The first is determining the left edge and right edge of the lost packet from the payload-cut packet. The second is how to represent the lost packet information such that it is compatible with SACK. For the first issue, we can easily obtain the left edge from the sequence number in the TCP header of the payload-cut packet. The original packet length in the IP header is preserved; hence, we can use it to calculate the right edge[4]. For the second issue, as shown in Fig. 4(b), we swap the position of the left and right edge to indicate lost packet information. This arrangement will not produce ambiguity because the left edge must be smaller than the right edge in the SACK option. In this manner, we can parsimoniously indicate lost packet information in the PACK option. When a payload-cut packet arrives, the receiver simply parses it and adds the lost packet information in the first block of the PACK option, which is similar to the processing in SACK. For convenience, the receiver marks the packet to tell the sender to parse the PACK option. Marking will be discussed in § 5.1.

For simplicity, the two-byte PACK-Permitted option is similar to SACK-Permitted option. The option is sent in the SYN by the sender to inform the receiver that it can support the PACK option. In our implementation, we just use the SACK-Permitted option: this does not affect the performance of CP.

4.4 Sender Reaction to PACK

TCP SACK maintains a "scoreboard" to store the status of packets [9]. Every packet is in one of four states, i.e., "received" (or "SACKed"), "out-of-order," "lost," and "retransmitted." Like SACK, CP stores packet information in a "scoreboard." When the new packet is transmitted, its status changes to "out-of-order." We divide the sender action on receiving PACK into three steps. In the first step, the sender parses the PACK option as described in § 4.3 and checks state of the packet. If the status of the packet is "received" or "lost," the sender takes no action. Otherwise, it enters the second step, converting the packet status in the scoreboard to "lost" and adding the packet to the retransmission queue. In the third step, if TCP is not in a state of fast retransmission, the sender triggers the fast retransmission mechanism and performs congestion control. When the packet has been retrans-

[4] right_edge = left_edge + SYN + FIN + original_length - IP_header_length - TCP_header_length. In Linux kernel, SYN and FIN have only 1 bytes payload by default, although the payload length is 0 in reality. Therefore, if the packet is SYN or FIN packet, SYN or FIN in the above formula equals 1. Otherwise, both are 0.

mitted, the packet status changes to "retransmitted". It should be noted that using three duplicate ACKs to indicate packet loss is not necessary unless payload-cut packets or PACKs themselves are lost. Therefore, we recommend that the threshold for duplicate ACKs is set to a relatively large number.

4.5 Benefits

CP overcomes the three TCP defects as follows:

Self-clocking Stall: Payload-cut packets and PACKs provide the sender with packet loss notification. Using this notification, the sender can maintain a self-clocking mechanism and avoid timeouts.

Inaccurate Packet loss Notification: Payload-cut packets provide the receiver with accurate packet-loss information, which is brought back to the sender by PACKs. Thus, the sender can easily distinguish between lost and out-of-order packets.

Slow Packet loss Detection: Because PACKs quickly carry the information of packet loss back to the sender, the sender could start the retransmission immediately without waiting for three duplicate ACKs. Therefore, CP shortens the packet loss detection time.

Note that CP is compatible with existing congestion control protocols proposed for data centers because it is an additional mechanism for loss detection. In addition, it is TCP-friendly and backward compatible.

4.6 Discussion

The extra buffer can be small: An extra buffer or a lower drop threshold is needed at a CP switch to hold payload-cut packets during an output buffer overload. It only needs to be one maximum packet size because the input bandwidth becomes smaller than output bandwidth after CP processing. For example, the average packet size (without MAC header) in DCNs is 850 bytes [8]. CP reduces the packet length to only 66 bytes, a reduction by about 92.37%[5]. Thus, with CP, a 1 Gbps output link can tolerate an input burst at about 13.1 Gbps (1 Gbps / 7.63%). Therefore, the extra buffer can be quite small without loss of payload-cut packets.

Loss of payload-cut packets or PACKs: The loss of payload-cut packets or PACKs is a rare occurrence because of their small size. Nevertheless, even if one PACK is dropped, the sender can recover the information of the lost PACK from a subsequent PACK. Similarly, if a payload-cut packet is dropped, the sender can use duplicate ACKs to confirm the lost packet, which causes CP to degenerate to standard TCP SACK.

[5]The average packet length is 864 B (14 MAC header + 850 average length) and payload-cut packet length is 66 B (14 MAC header + 20 IP header + 20 TCP header + 12 timestamp option). The ratio of the header length to packet length is about 7.63% for this packet length.

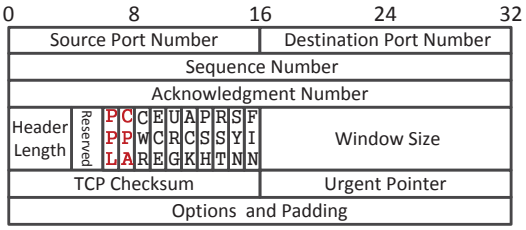


Figure 5: Extended TCP header for CP implementation

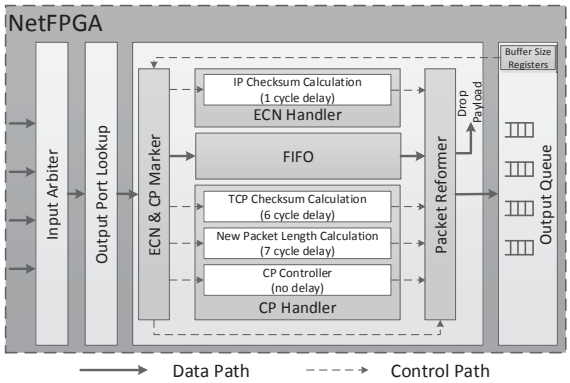


Figure 6: Structure of NetFPGA for implementing the CP switch. The packets are forwarded through the data path, and the decision-making modules in the data path generate signals that tell the processing modules whether or how to deal with the packet via the control path.

5 Implementation

This section discusses CP implementation details.

5.1 Protocol Details

As shown in Fig. 5, two reserved bits in the TCP header are defined as CP-available (CPA) and precise packet loss (PPL) to identify the four types of packets mentioned in § 4.1. Packets with CPA = 1 and PPL = 0 are CP-enabled packets and those with CPA = 0 and PPL = 1 are PACK packets. When the payload of CP-enabled packets are cut off, PPL becomes 1. Therefore, packets with CPA = 1 and PPL = 1 are payload-cut packets. To maintain compatibility, packets with CPA = 0 and PPL = 0 are defined as normal TCP packets. CP switches handle them in the same way as commodity switches.

5.2 CP switch

CP switches cut off the payload of CP-enabled packets and change the TCP checksum when a buffer exceeds the threshold. In our prototype, we also add ECN capability to allow us to compare DCTCP with CP as discussed in § 6, .

Fig. 6 shows the basic structure of the CP switch implemented in NetFPGA cards. When a packet enters the switch, it first looks for an output port and then waits for the handling of the ECN & CP marker. Depending on the packet type and buffer occupancy level, the ECN & CP marker determines how to deal with packets

Table 1: Resource Usage of NetFPGA

	Reference Switch	ECN Switch	CP Switch
Slices	12807	13579	13777
Slice Flip Flops	14158	14365	14511
4 Input LUTs	17589	17907	18239

and directs the operation of related modules. For example, on seeing payload-cut packets or PACK packets, the ECN & CP marker allows them to pass directly through all modules in the data path and places them in the forwarding buffer. All modules work in parallel, and the packet waits in the FIFO queue until all modules have finished their processing. After a short processing delay (seven-cycle delay for CP handler; one-cycle delay for ECN handler; no delay for others[6]), packet reformer regenerates the packet, which introduces no delay and little additional overhead. Finally, the packet is placed in the output buffer and waits for forwarding. As discussed in § 4.6, an extra buffer space may be necessary to absorb all payload-cut packets corresponding to dropped packets; however, the extra buffer size can generally be quite small, because some payload-cut packet loss is acceptable. Therefore, the extra buffer is set to 4 KB in our implementation.

The CP handler comes out three operations to implement CP: **(1) TCP checksum calculation.** Recomputing a checksum is only needed for the TCP header whose maximum size is 48 B (8 IP address + 20 TCP header + 20 padding); thus, the overhead is very limited. For example, NetFPGA can process 8 B in a cycle, and no more than 6 cycles (i.e., 48 ns) are consumed to re-compute the checksum. With increased processing capacity, this overhead can be further reduced and will become negligible. **(2) New packet length calculation.** This module changes the packet length parameter when sending a packet header because of NetFPGA requirement. **(3) CP controller.** This module tells packet reformer to cut out the payload.

Clearly, CP implementation is quite simple, and the processing delay at the switch is very small. Furthermore, the cutting off payload mechanism is not a normal operation for each packet, because it is only triggered to avoid packet dropping. Therefore, the additional processing delay at the switch is not introduced frequently. In addition, CP introduces only a little resource consumption on switches. Table 1 shows that, compared with ECN switches, the resource usage in CP switches only increases by less than 2%.

[6]The clock rate of NetFPGA is 125 MHz. Each cycle is equivalent to 8 ns, i.e., CP drop processing introduces a delay of 56 ns, and an additional delay of ECN is 8 ns. ECN marking must change one bit in each packet; thus, the new IP checksum is similar to the former and is easily calculated. Otherwise, the CP mechanism will cut off all payloads and require more time to recalculate the TCP checksum.

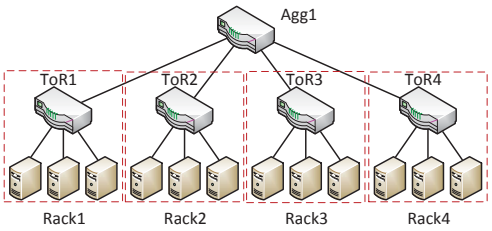


Figure 7: Basic topology of our testbed

We conclude that the implementation complexity, processing delay, and resource consumption of CP switches are acceptable; thus, CP drop processing can be built into commercial switches.

6 Evaluation

This section is divided into four parts. In each subsection, we test the effectiveness of CP in addressing each TCP problem mentioned in § 2. We also compare its performance with that of other state-of-the-art protocols used in DCNs. Our experimental results showed that CP addresses the TCP problems and achieves the expected performance goals.

Testbed. We use a real testbed to evaluate CP performance. Our testbed is shown in Fig. 7. All switches in our experiments are NetFPGA cards with four 1-Gbps Ethernet ports. The implementation of the CP switch as described in § 5.2 is downloaded to the NetFPGA cards, and the buffer sizes of each port are arbitrarily set from 1 KB to 512 KB. Each ToR switch communicates with others through the aggregation switch and connects three hosts (Dell OptiPlex 360 desktops with an Intel Celeron Dual-Core 2930 MHz CPU, 4 GB RAM and 1 Gbps NIC). All hosts in our testbed are running CentOS 5.5 with Linux kernel 2.6.38 with protocol patches applied. The RTT without queuing delay is approximately 100 μs between two endhosts in the same rack.

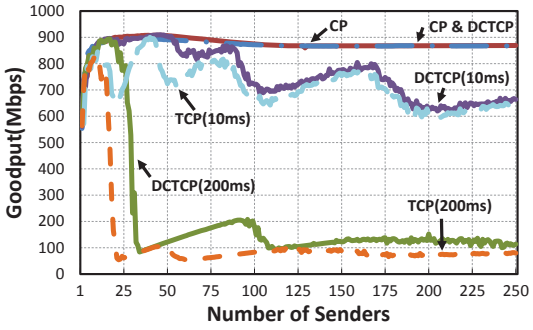
Protocols. We study four congestion control schemes.

(i) TCP: The TCP variant we study is TCP SACK. We also allow DSACK [13]. The TCP receive window size is set to 256 KB so that TCP can meet a 1 Gbps line rate. We disable delayed ACK to avoid performance problem [30]. The number of tolerable out-of-order packets is three by default. Furthermore, RTO_{min} is set to either 10 ms or 200 ms and this is denoted TCP (10ms) and TCP (200ms), respectively.

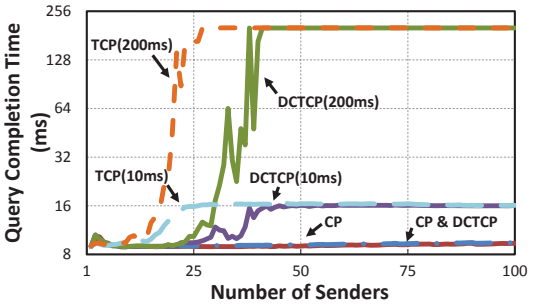
(ii) CP: Switches carry out CP; receivers generate PACKs. Other settings are the same as TCP.

(iii) DCTCP: The parameters are set to $K = 32$ KB and $g = 1/16$ [4]. The other settings are the same as for TCP, including two values for RTO_{min}.

(iv) DCTCP with CP (CP&DCTCP): CP is used along with DCTCP. Other settings are the same as for DCTCP.



(a) MapReduce-like Apps



(b) Web search Apps

Figure 8: Experimental results of scenarios for many-to-one transmission. Notice the log scale in Fig. 8(b).

6.1 Low and Volatile Throughput Impairment

Here, we explore the results of using CP in two typical scenarios (MapReduce-like and Web search application scenarios), which have different performance criteria (see § 2.1).

Topology and parameter settings. In these tests, nine hosts send data to a host on another rack. Each sending host is used to emulate multiple senders [28]. Flows are bottlenecked at the link from the aggregation switch to the receiving host. The buffer on the bottleneck link is 128 KB, and the other buffers are 512 KB. Because the settings of RTO_{min} can have a significant impact on performance in these scenarios [27], two different RTO_{min} settings (10ms and 200ms) were studied.

MapReduce-like application scenario. In this test, a receiver generates a query to each sender, and each of them immediately responds with 64 KB of data. Fig. 8(a) shows that only CP successfully avoids both TCP Incast and the throughput cliff when the number of senders is large. both DCTCP and TCP with 10-ms-RTO_{min} maintain a throughput of about 600 Mbps; however, the throughput cliff is not avoided. We see that CP and DCTCP with CP achieve a high throughput[7] and avoid throughput cliff because the payload-cut packets and PACK packets maintain the self-clocking.

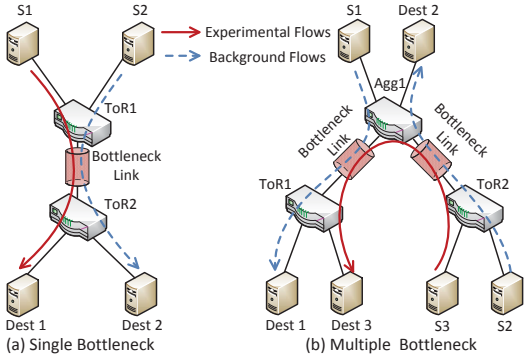


Figure 9: Experimental scenario of unfairness impairment assessment.

Web search application scenario. In this test, each of n different senders transfers $1024/n$ KB to the receiver through the bottleneck link. Fig. 8(b) shows that CP and DCTCP with CP have better performance than TCP and DCTCP in reducing query completion time. The query completion times of CP and DCTCP with CP are approximately 9 ms when the number of senders is less than 50 and slightly increases by 0.4 ms as the number of senders increases from 1 to 100. In comparison, depending on 10-ms- and 200-ms-RTO_{min}, the delays of TCP or DCTCP converge at 16.1 ms and 201.6 ms, respectively. Through closer observation and analysis, we found that CP totally eliminates timeouts and wastes only 3.49% of the 1 Gbps bandwidth to send 554 payload-cut packets (554×66 B = 35.7 KB) for each query when the number of senders is 100. However, increasing the number of senders[8] for TCP and DCTCP results in timeouts and prolongs the query completion time. In addition, using DCTCP with CP reduces the query completion time by 91.3 μs compared with CP because DCTCP with CP maintains short queue length and reduces the probability of packet loss.

6.2 TCP Unfairness

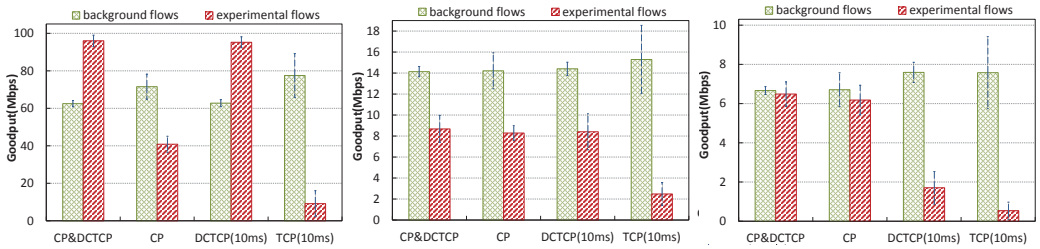
In this section, we focus on whether CP can improve TCP unfairness.

Topology. We use topologies studied in prior works for both the single-bottleneck case [24] and the multiple-bottleneck case [20], as shown in Fig. 9. It should be noted that the number of background flows shown in Fig. 9 (b) are equal. We use a subset of our testbed to achieve these scenarios, and each buffer on the bottleneck links is 128 KB[9].

[7] In CP implementation, the receiver obtains approximately 43 payload-cut packets for each sender and only wastes 4.32% of the 1 Gbps bandwidth when the number of senders is 250.

[8] In our experiment, DCTCP and TCP suffer from TCP Incast when the numbers of senders are 38 and 23, respectively. Below these number of senders, there are large fluctuations in the query completion time even if TCP Incast disappears.

[9] only the critical components are illustrated. An aggregation switch (Fig. 9(a)) and a ToR switch (Fig. 9(b)) are not shown because they do not affect the results.



(a) 12-2 flow pair (b) 60-10 flow pair (c) 120-20 flow pair

Figure 10: Average goodput of different flow-pairs in the single-bottleneck experiment. We use the notation $n-m$ to refer to n background and m experimental flows.

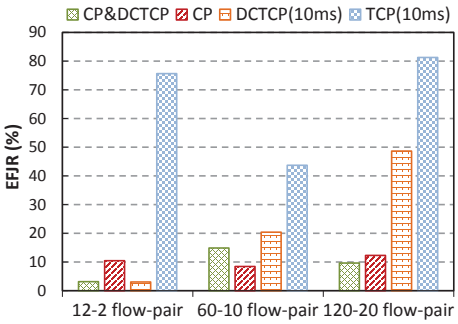
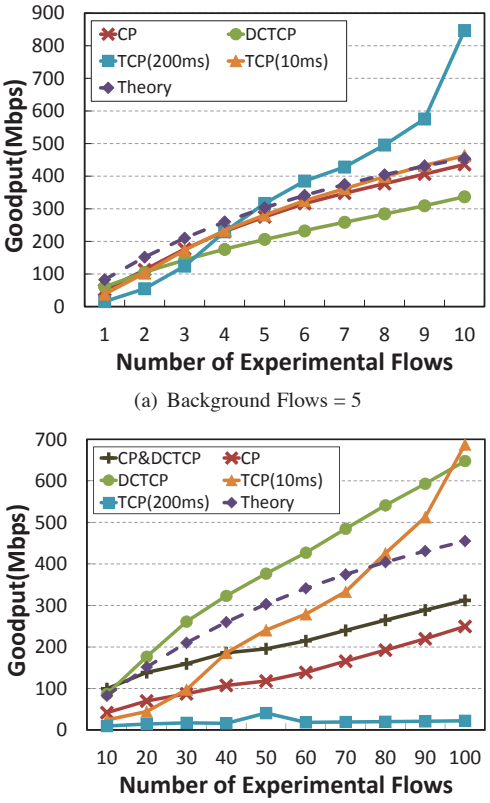


Figure 11: Experimental flows jitter ratio (EFJR) in single-bottleneck scenario

Single-bottleneck scenario. We refer to the combination of n background flows and m experimental flows as an n-m flow pair. From the experiments, we found that TCP and DCTCP with different RTO_{min} have similar features; therefore, we only show the results for 10-ms-RTO_{min} TCP and DCTCP. Three flow pairs from our experiment are shown in Fig. 10. The error bars indicate the average absolute deviation of goodput. For convenience, we refer to the experimental flow ratio of the average absolute deviation to average throughput as the experimental flow jitter ratio (EFJR). EFJR reflects the fairness among experimental flows.

Three points are evident from Fig. 10 and Fig. 11. First, the experimental flows using TCP suffer the TCP Outcast problem in all three cases. The goodput of flows with larger number is much lower than that with smaller number. Second, DCTCP can alleviate the TCP Outcast problem for the 60-10 flow pair and the 12-2 flow pair; however, when the number of flows is large (e.g., the 120-20 flow pair), DCTCP suffers the TCP Outcast problem and its EFJR is high. In additional, the EFJR of DCTCP significantly fluctuates from 3% to 48%. It is evident that DCTCP can maintain flow fairness only when number of flows is small. Third, CP prevents TCP Outcast from occurring and maintains the EFJR below 15%. DCTCP with CP has a smaller EFJR than CP; therefore, DCTCP with CP provides better fairness among flows. In conclusion, compared with TCP and DCTCP, DCTCP with CP and CP both avoid TCP Outcast problem and achieve reasonable fairness among flows.



(a) Background Flows = 5

(b) Background Flows = 50

Figure 12: Total goodput of all experimental flows in the multiple-bottleneck scenario.

Multiple-bottleneck scenario. In this scenario, from Fig. 9(b), we see that experimental flows are MBFs and background flows are SBFs. Two sets of experiments were conducted to explore the fairness of experimental flows under different packet loss conditions. We chose 5 flows and 50 flows for the number of background flows to represent low and high packet loss scenarios, respectively. Each experiment lasted 22 minutes. The ratio of the number of experimental flows to that of the background flows was increased by 20% every two minutes from 0 to 200%. From the experiments, we found that DCTCP with different RTO_{min} values has a similar per-

formance, so Fig. 12 only illustrates DCTCP with 200-ms-RTO$_{min}$, denoted DCTCP.

In Fig. 12, the line denoted "Theory" is determined by calculating the throughput according to RTT fairness [10]. Deviation above this line indicates unfairness to SBFs, and below this line to MBFs. Three conclusions can be drawn from the data presented in the low loss rate experiment shown in Fig. 12(a). First, we discover that 200-ms-RTO$_{min}$ TCP suffers the TCP Outcast problem when the number of flows is greater than 8 because background flows suffers timeout and share extremely low bandwidth. Under these conditions, the other protocols show normal performance. Second, compared with 10-ms-RTO$_{min}$ TCP and CP, DCTCP demonstrates poor fairness because it only achieves an average of 70% of the theoretical fair value, while 10-ms-RTO$_{min}$ TCP and CP achieve an average of 87.1% and 86.8%, respectively. Third, compared to 10-ms-RTO$_{min}$ TCP, CP has slight unfairness because rapid packet loss notification favors short RTT flows. Fig. 12(b) shows that, in a high packet loss environment, 200-ms-RTO$_{min}$ TCP has very low goodput. In addition, 10-ms-RTO$_{min}$ TCP and DCTCP experience the TCP Outcast problem when the number of experimental flows increases. In conclusion, CP and DCTCP with CP perform better.

From these results above, we conclude that, as the packet loss ratio increases, the TCP Outcast problem occurs initially and then low throughput of MBF occurs. Both phenomena affect TCP fairness; 10-ms-RTO$_{min}$ TCP and DCTCP can only alleviate the problem but not solve it. Furthermore, DCTCP with CP significantly improves TCP fairness and increases throughput up to 75.1% of the theoretical fair value. Unfortunately, CP alone, which achieves only an average of only 45.6% of the theoretical value, only eases unfairness rather than solving it because TCP congestion control is significantly affected by the violent queue oscillations in a high packet loss environment. Thus, CP with DCTCP is the best approach to maintain fairness in low or high packet loss environments with multiple bottlenecks.

6.3 TCP Out-of-order Problem

Topology. Fig. 13 shows the basic topology used in these experiments. An additional aggregation switch is added to the testbed. It should be noted that the ToR1 switch is a random forwarding switch that sends packets from S1 to Path A or Path B with the probability of selecting each path chosen by a configuration parameter. Other switches forward packets according to the look-up table.

[10] We define G as the theoretical goodput and N as the number of flows. Through experimental measurements, we found that the RTT of background flows is half that of experimental flows. Therefore, $G_{experimental} = N_{experimental}/(N_{background} * 2 + N_{experimental}) \times G_{total}$ where $G_{total} = 910Mbps$.

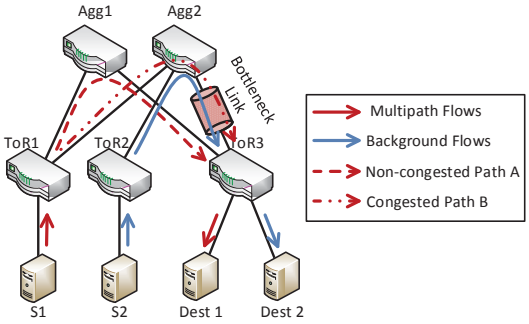


Figure 13: Basic topology of TCP out-of-order experiment

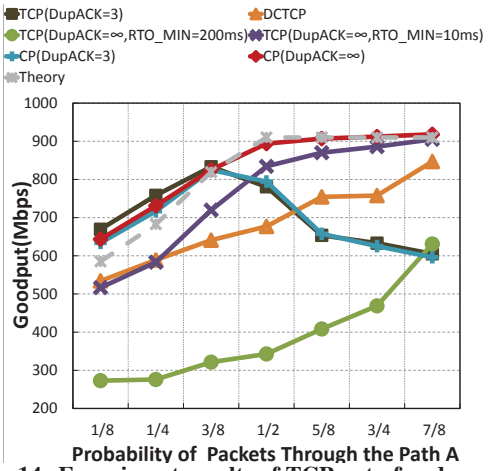


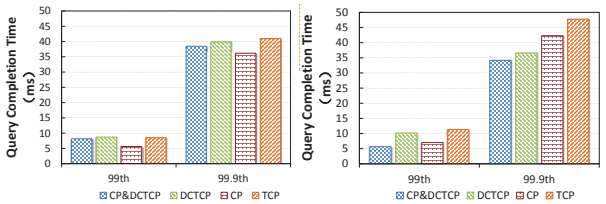
Figure 14: Experiment results of TCP out-of-order experiment

ACKs from Dest1 only pass through Path B back to S1. Furthermore, the bottleneck buffer is set to 256 KB. This creates an environment in which packets passing through Path A to reach Dest1 take less time than the packets through Path B.

Experimental parameter settings. We allow the number of tolerable out-of-order packets to be set to three or infinity. In our experiment, DCTCP achieved similar throughput in both cases; thus, we only show the results for DCTCP with three duplicate ACKs (denoted DCTCP in Fig. 14). The theoretical value is calculated by the probability of packets passing through Path A [11].

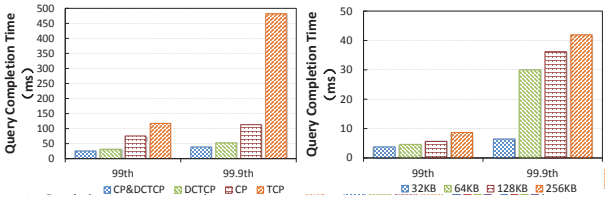
Experimental results. We can draw four conclusions from the results shown in Fig. 14. First, the small number of tolerable out-of-order packets (three) causes throughput decline even though most packets pass through the non-congested Path A. With the increasing number of packets passing through Path A, the throughput of TCP or CP with three duplicate ACKs decreases from 825 Mbps to around 600 Mbps. This occurs because a small number of packets moving through the congested Path B leads to spurious retransmission and

[11] We define G as the theoretical goodput, P as the probability of packets through the Path A. Theoretically the sub-flows of multi-path flows through Path B gets half of the bandwidth. Therefore, $G_{Multipath} = 1/2 \times G_{bandwith}/(1-P)$ where $G_{bandwith} = 910Mbps$.



(a) Agg Buffer=128KB (b) Agg Buffer=384KB

Figure 15: Query completion time under realistic Data Center traffic



(a) 10X Expanded Traffic (b) Different Buffer Size

Figure 16: Query completion time under different conditions.

unnecessary congestion control. Second, the large number of tolerable out-of-order packets (infinity) causes TCP to have a slow response to congestion and triggers a significant number of timeouts. From the comparison of 200-ms-RTO$_{min}$ TCP, 10-ms-RTO$_{min}$ TCP and the theoretical value, we find that 200-ms-RTO$_{min}$ TCP achieves less throughput than 10-ms-RTO$_{min}$ TCP because RTO$_{min}$ determines how quickly the protocol reacts to congestion as the timeout occurs. Rapid reaction to congestion increases total bandwidth utilization. In addition, the extent of congestion declines with increasing probability of packets passing through Path A. Therefore, TCP throughput approaches the theoretical value. Third, DCTCP is affected by the ECN mechanism and the early congestion control makes the throughput lower than the theoretical value. Finally, CP with no threshold for out-of-order packets works well and matches the theoretical value. In summary, CP with no threshold for duplicate ACKs can completely solve the out-of-order problem.

6.4 Query Completion Time

Experimental parameter settings. The literature [4] describes the PDF of background flow size distribution, the interval time between arrivals of queries, and the interval time between arrivals of background flows in realistic DCNs. According to this information, we generate realistic traffic of DCNs with 12 servers in our testbed. Unless otherwise specified, all switch buffers of each port are set to 128 KB. In the experiment, each server independently selects a time value and data size from identical time interval and data size distributions. One query is immediately sent to the other 11 servers after its arrival, and responses are sent back to the originating server. Both query size and response size are 2 KB. We conducted the experiment using DCTCP with CP, CP, 10-ms-RTO$_{min}$ DCTCP, and 10-ms-RTO$_{min}$ TCP. The experiment lasted 10 minutes and generated over 50,000 queries and background flows separately. We also conducted the previously reported 10x-realistic traffic experiment [4] in which the size of responses and background flows larger than 1 MB were increased tenfold. Furthermore, in the different buffer size experiments, all switch buffers are set to a specific value.

Realistic data center traffic. Fig. 15 shows the 99th and 99.9th percentile of query completion time with 128 KB and 384 KB at the aggregation switch. It can be seen that CP reduces the query completion time of TCP and DCTCP. In Fig. 15(a), compared with TCP, CP achieves a 34.06% reduction at 99th percentile and a 11.74% reduction at 99.9th percentile, respectively. At the 99th and 99.9th percentile, query completion time of DCTCP is 0.56ms and 1.41ms higher than that of DCTCP with CP, respectively. Similarly, in Fig. 15(b), query completion time of CP is 61.91% and 88.60% of that of TCP at the 99th and 99.9th percentile, respectively. Compared with DCTCP, CP with DCTCP achieves a 4.5ms decrease at 99th percentile and a 2.4ms decrease at 99.9th percentile. No query traffic suffered timeouts during the experiments. Thus, the query completion time reduction is due to the rapid packet loss detection using CP.

Expanded traffic or different buffer size. We conducted experiments using 10x-expanded traffic and a different buffer size. As can be seen in Fig. 16(a), compared with TCP, CP achieves a 35.71% reduction at 99th percentile and a 76.55% reduction at 99.9th percentile. Compared with DCTCP, DCTCP with CP achieves a 18.5% reduction at 99th percentile and a 26.2% reduction at 99.9th percentile. These results indicate that CP effectively decreases query completion time even under 10x-expanded traffic condition.

Fig. 16(b) shows the query completion times with CP with different buffer size. Compared with 128 KB and 256 KB buffer, 32 KB buffer achieves 33.07% and 56.49% time reduction at the 99th percentile, respectively. The 99.9th percentile query completion time with a 32 KB buffer is only 6.426 ms, which is 17.77% and 15.34% of that with 128 KB and 256 KB buffer, respectively. It is clear that the combination of CP and a shallow buffer can achieve good performance in DCNs.

7 Conclusion

In this paper, we proposed the cutting payload (CP) approach to solve TCP problems. Analysis indicates that TCP problems are due to three types of issues. To address these imperfections, CP uses payload-cut packets and PACK packets to maintain self-clocking and to

rapidly and precisely inform a sender of packet loss. The experimental results verify that CP can solve the problems with TCP discussed in this paper. In addition, CP is compatible with nearly all existing congestion control protocols in DCNs. In particular, the combination of DCTCP and CP has the best performance across all topologies in experiments.

Acknowledgments

We gratefully appreciate our shepherd Prof. S. Keshav for his constructive suggestions, and acknowledge the anonymous reviewers for their valuable comments. This work is supported in part by National Basic Research Program of China (973 Program) under Grant No. 2012CB315803 and 2014CB347800, and National Natural Science Foundation of China (NSFC) under Grant No. 61225011.

References

[1] M. Al-Fares, A. Loukissas, and A. Vahdat. A Scalable, Commodity Data Center Network Architecture. In *SIGCOMM*, August 2010.

[2] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: Dynamic Flow Scheduling for Data Center Networks. In *NSDI*, August 2010.

[3] M. Alizadeh, B. Atikoglu, A. Kabbani, A. Lakshmikantha, R. Pan, B. Prabhakar, and M. Seaman. Data Center Transport Mechanisms: Congestion Control Theory and IEEE Standardization. In *Allerton CCC*, August 2008.

[4] M. Alizadeh, A. Greenberg, D. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. DCTCP: Efficient Packet Transport for the Commoditized Data Center. In *SIGCOMM*, August 2010.

[5] M. Alizadeh, A. Kabbani, T. Edsall, B. Prabhakar, A. Vahdat, and M. Yasuda. Less Is More: Trading a Little Bandwidth for Ultra-Low Latency in the Data Center. In *NSDI*, April 2012.

[6] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker. pFabric: Minimal Near-Optimal Datacenter Transport. In *SIGCOMM*, August 2013.

[7] M. Allman, H. Balakrishnan, and S. Floyd. RFC 3042: Enhancing TCP's Loss Recovery Using Limited Transmit.

[8] T. Benson, A. Anand, A. Akella, and M. Zhang. Understanding Data Center Traffic Characteristics. In *WREN*, August 2009.

[9] E. Blanton, M. Allman, K. Fall, and L. Wang. RFC 3517: A Conservative Selective Acknowledgment (SACK)-based Loss Recovery Algorithm for TCP.

[10] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, December 2004.

[11] A. Dixit, P. Prakash, and R. R. Kompella. On the Efficacy of Fine-grained Traffic Splitting Protocols in Data Center Networks. In *SIGCOMM*, August 2011.

[12] N. Dukkipati. Tcp: Tail Loss Probe (TLP). `http://lwn.net/Articles/542642/`.

[13] S. Floyd, J. Mahdavi, M. Mathis, and M. Podolsky. RFC 2883: An Extension to the Selective Acknowledgement (SACK) Option for TCP.

[14] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2: A Scalable and Flexible Data Center Network. In *SIGCOMM*, August 2009.

[15] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S. Lu. Bcube: A High Performance, Server-centric Network Architecture for Modular Data Centers. In *SIGCOMM*, August 2009.

[16] C. Hong, M. Caesar, and P. B. Godfrey. Finishing Flows Quickly with Preemptive Scheduling. In *SIGCOMM*, August 2012.

[17] V. Jacobson. Congestion Avoidance and Control. In *SIGCOMM*, August 1988.

[18] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken. The Nature of Datacenter Traffic: Measurements & Analysis. In *IMC*, November 2009.

[19] S. Kavulya, J. Tan, R. Gandhi, and P. Narasimhan. An Analysis of Traces from a Production Mapreduce Cluster. In *CCGRID*, May 2010.

[20] A. Mankin. Random Drop Congestion Control. In *SIGCOMM*, September 1990.

[21] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. RFC 2018: TCP Selective Acknowledgment Options.

[22] D. Nagle, D. Serenyi, and A. Matthews. The Panasas ActiveScale Storage Cluster: Delivering scalable high bandwidth storage. In *SC*, November 2004.

[23] J. Postel. RFC 792: Internet Control Message Protocol.

[24] P. Prakash, A. Dixit, Y. C. Hu, and R. Kompella. The TCP Outcast Problem: Exposing Unfairness in Data Center Networks. In *NSDI*, April 2012.

[25] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley. Improving Datacenter Performance and Robustness with Multipath TCP. In *SIGCOMM*, August 2011.

[26] B. Vamanan, J. Hasan, and T. N. Vijaykumar. Deadline-Aware Datacenter TCP(D^{2}TCP). In *SIGCOMM*, August 2012.

[27] V. Vasudevan, A. Phanishayee, H. Shah, E. Krevat, D. G. Andersen, G. R. Ganger, G. A. Gibson, and B. Mueller. Safe and Effective Fine-grained TCP Retransmissions for Datacenter Communication. In *SIGCOMM*, August 2009.

[28] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowstron. Better Never than Late: Meeting Deadlines in Datacenter Networks. In *SIGCOMM*, August 2011.

[29] H. Wu, Z. Feng, C. Guo, and Y. Zhang. ICTCP: Incast Congestion Control for TCP in Data Center Networks. In *CoNEXT*, December 2010.

[30] M. Yu, A. Greenberg, D. Maltz, J. Rexford, L. Yuan, S. Kandula, and C. Kim. Profiling Network Performance for Multi-tier Data Center Applications. In *NSDI*, March 2011.

[31] L. Zhang. Why TCP Timers Don't Work Well. In *SIGCOMM*, August 1986.

[32] M. Zhang, B. Karp, S. Floyd, and L. Peterson. RR-TCP: A Reordering-Robust TCP with DSACK. In *Proceedings of ICNP*, November 2003.