Slowing Little Quickens More: Improving DCTCP for Massive Concurrent Flows

Mao Miao, Peng Cheng, Fengyuan Ren, Ran Shu
Department of Computer Science and Technology, Tsinghua University, China
Tsinghua National Laboratory for Information Science and Technology, China
{miaomao, chengpeng5555, renfy, shur11}@csnet1.cs.tsinghua.edu.cn

***Abstract*—DCTCP is a potential TCP replacement to satisfy the requirements of data center network. It receives wide concerns in both academic and industrial circles. However, DCTCP could only support tens of concurrent flows well and suffers timeouts and throughput collapse facing numerous concurrent flows. This is far from the requirement of data center network. Data centers employing partition/aggregation pattern usually involve hundreds of concurrent flows. In this paper, after tracing DCTCP's dynamic behavior through experiments, we explored two roots for DCTCP's failure under the high fan-in traffic pattern: (1) The regulation mechanism of sending window is ineffective when cwnd is decreased to the minimum size; (2) The bursts induced by synchronized flows with small cwnd cause fatal packet loss leading to severe timeouts. We enhance DCTCP to support massive concurrent flows by regulating the sending time interval and desynchronizing the sending time in particular conditions. The new protocol called DCTCP$^+$ outperforms DCTCP when the number of concurrent flows increases to several hundreds. DCTCP$^+$ can normally work to effectively support the short concurrent query responses in the benchmark from real production clusters, and keep the same good performance with the mixture of background traffic.**

***Keywords*— DCTCP; Timeout; Massive concurrent flows; Throughput; Latency**

I. Introduction

Today's data centers have been hosting more and more diverse applications, mixing workloads with different requirements, including low predictable latency and large sustained throughput. In this environment, traditional TCP protocol falls short. DCTCP [1] is proposed to satisfy the requirements for data center's applications: low latency for short flows, high tolerance for micro burst traffic and high utilization for long flows. Due to the effectiveness in congestion control and the simplicity and compatibility for deployment, DCTCP owns its dominant influence in both academic and industrial circles. Subsequent work, such as D^{2}TCP protocol [2], the High-bandwidth Ultra-Low (HULL) architecture [3] and Data Center Congestion Control [4], are all built on the basis of DCTCP. Microsoft also claimed that Windows Server 2012 had supported DCTCP to deal with the network congestion in a more intelligent way [5].

Although DCTCP contributes significant performance improvement, it still has nonnegligible limitations. As claimed in the original paper [1], DCTCP can only support tens of concurrent flows well when commodity switches employing static shared buffer are deployed in data centers. But it suffers from severe timeouts and throughput collapse as massive concurrent flows are active, which significantly extends the flow completion time (FCT). However, the number of concurrent flows in typical online services employing the divide and conquer computing paradigm is far greater than that DCTCP can support well with. For example, Yahoo!'s M45 supercomputing MapReduce cluster [6] parallelizes and distributes jobs across large clusters with each job consisting of hundreds of Maps on average; Google web search [2][7] and Microsoft Bing [8] are reported the interactive service processing consists of parallelization across 10s-1000s of servers and aggregation of responses across network. Many measurements on productive data centers also confirm the high fan-in traffic pattern [9][10]. Therefore, supporting massive concurrent flows is necessary in today's data center network. Unfortunately, DCTCP is unable to work well in this situation. In this paper, we focus on remedying the pitfall of DCTCP. The main contributions are two-fold.

First, we trace and examine DCTCP's dynamic behavior at senders in real experiments, identify the radical reasons for DCTCP's failure under the high fan-in traffic pattern: (1) DCTCP is unable to further adjust the sending rate when the congestion window (`cwnd`) of some flows reaches to the minimum size despite receiving the ECN feedback yet. (2) The bursts caused by the synchronization of concurrent flows with small `cwnd` lead to the fatal packet loss which directly triggers timeouts, since the pipeline capacity in data center network is relatively small. N synchronized concurrent flows with the minor `cwnd` value could easily overflow the buffer associated with the bottleneck link, resulting in the awful *Full Window Loss Timeout* and *Lack of ACKs Timeout* [11][12].

Second, in the light of reasoning, we propose DCTCP$^+$ to support massive concurrent flows. The core idea of the enhancement mechanism employed by DCTCP$^+$ is straightforward. When `cwnd` reaches to the minimum size, and the sender is required to further decrease its `cwnd`, DCTCP$^+$ will regulate the sending time interval to slow down its sending rate. Furthermore, to avoid fatal packet losses caused by the synchronized concurrent flows, the sending time interval is randomized. Despite these additional mechanisms increase FCT by hundreds of microseconds, comparing to the worse timeouts, it is worth to slow down sending packets since it shortens FCT on the whole.

We demonstrate a full implementation of DCTCP$^+$ on our testbed. DCTCP$^+$ needs less than 100 lines change to DCTCP. The performance of DCTCP$^+$ is evaluated under different traffic scenarios, compared with DCTCP and TCP. The results show that in the basic incast experiment, DCTCP$^+$ keeps relatively high good throughput and low completion time. When the number of concurrent flows increases to several hundreds, DCTCP$^+$ also outperforms DCTCP in the benchmark traffic following the statistical features from production clusters.

The rest of the paper is structured as follow. Section II introduces the background. Section III presents the experimental configurations. Section IV shows the detailed dynamic behavior of DCTCP under the high fan-in traffic pattern, explores the radical reasons of DCTCP's failure. Section V describes the design and implementation of DCTCP$^+$. The experimental results and evaluation are presented in Section VI. Some issues about extension of DCTCP$^+$ are discussed in Section VII. Section VIII talks about the related work. Finally, the paper is concluded in Section IX.

II. Background

In this section, we briefly introduce DCTCP protocol and then present two common and significant features of data center network: the high fan-in traffic pattern and the small pipeline capacity.

A. Data Center TCP (DCTCP)

DCTCP [1] is proposed to address the shortcomings of TCP in data center networks. On the switch side, it employs a simple queue management scheme based on Explicit Congestion Notification (ECN). The switch sets the ECN bit for all the incoming packets once the queue length exceeds the reference buffer threshold K. On the source side, compared with the standard TCP's responding on ECN mark by halving its window size, DCTCP adjusts its sending window size following Equation (1) and (2), where F is the percentage of packets marked ECN bit which reflects the extent of congestion at the bottleneck link, g is a fixed parameter.

$$\alpha \leftarrow (1-g)\alpha + \alpha F, \quad g \in (0,1) \tag{1}$$

$$W \leftarrow (1-\frac{\alpha}{2})W, \quad W \in [2, rwnd] \tag{2}$$

DCTCP can deal with the congestion well based on the ECN feedback, and can keep the switch queue length oscillating around threshold K under normal conditions, which is beneficial for improving burst tolerance and reducing end-to-end latency. These features are potential for DCTCP to be widely deployed in data centers comprising of low-cost, shared and shallow-buffer commodity switches.

B. High Fan-in Traffic Pattern

The high fan-in concurrent flows is a prominent feature for data centers recently, as the divide-and-conquer computing paradigm is widely used in the online and data-intensive applications [2][8]. For example, the user query is usually partitioned to numerous leaf nodes, and the results are aggregated by parent nodes recursively. Since all leaf nodes receive queries nearly the same time and respond almost simultaneously, it results in a high fan-in traffic in networks. Considering background traffic containing long flows, which consumes some available buffer, the buffer is easily overwhelmed to cause instaneous congestions.

Some measurement results confirm this feature. Yahoo!'s M45 MapReduce cluster [6][13] is reported each job consists of average 153 Maps and 19 Reduces. Considering the parallelized data processing of MapReduce framework [14], hundreds of concurrent flows share the same bottleneck link. Google web search cluster employs every query operating on data spanning thousands of servers, where a single query reads hundreds of megabytes on average [2][7]. Microsoft Bing project team also claims that the interactive service processing consists of parallelization across 10s-1000s of servers and aggregation of responses across networks [8].

C. Small Pipeline Capacity

The small pipeline capacity is another feature for today's data center network. We define the *Pipeline Capacity* as $C \times D + B$ where C is the bottleneck link capacity, D is the RTT delay, and B is the size of buffer associated with the bottleneck link. The RTT of data center network is usually 10s to 100s of microseconds [13][14-16]. Obviously, the in-flight capacity, $C \times D$, is quite small compared with the Wide Area Network where RTT is usually hundreds of milliseconds. What's more, low-cost commodity switches with buffer size of hundreds of kilobytes are widely deployed in data centers. This is because increasing switch buffer can extend the queuing delay and also means a substantial cost—switches with 1MB packet buffering per port may cost as much as $500,000 [15]. Therefore, the tiny $C \times D$ plus the shallow B results in the small *Pipeline Capacity* for data center network. It makes the network a low burst tolerance capability, which easily arouse packet dropping and degrades the transmission performance in data center networks.

III. Experimental Configuration

In order to clearly observe the detailed dynamic behavior of DCTCP facing the high fan-in burst traffic in data center network with small *Pipeline Capacity*, we conduct the experiments in our testbed, on which we could control the number of concurrent flows and monitor the detailed behavior of protocol stacks. The cluster contains 10 Dell OptiPlex 780 servers, connected through the NetFPGA-implemented GbE switches. The ECN-enabled switches owns four 1Gbps ports. The server is equipped with Intel Celeron Dual-Core 2930MHz CPU, 4GB RAM and 1Gbps NIC with the CentOS 5.5 distribution installed.

Our topology is a canonical tree-based 2-Tier topology. The benchmark is referred to the source code [1] [16]. Considering the large number of concurrent senders in experiments, and the limited number of servers, we implement a multithreading version to make each server maintain specific number of

[1] Available at: https://github.com/amarp/incast.

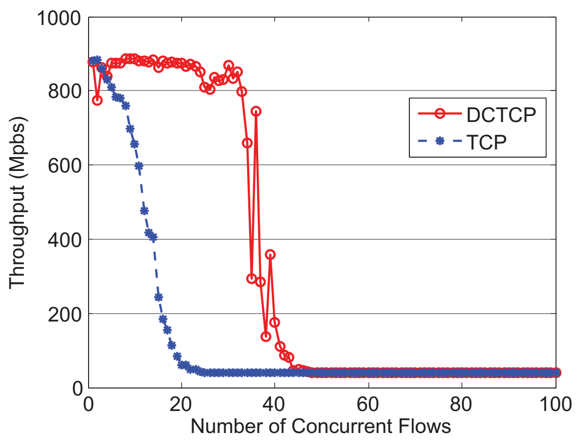


Fig. 1. Goodput of DCTCP and TCP with the increasing number of concurrent flows

concurrent flows, instead of each server sender opening one flow. Under this experimental configuration, we redo the same experiments in [1] for the large number of concurrent flows and get the similar results. We use `tcp_probe` kernel module by modifying `Kprobes`[17] to monitor the in-kernel variables of the protocol stack.

IV. Problems of DCTCP and analysis in detail

In this section, we introduce the detail about our observation on DCTCP's dynamic behavior in high fan-in traffic, present our analysis about why DCTCP fails in handling this traffic pattern and figure out the radical reasons of the problems that provide guidelines towards our design.

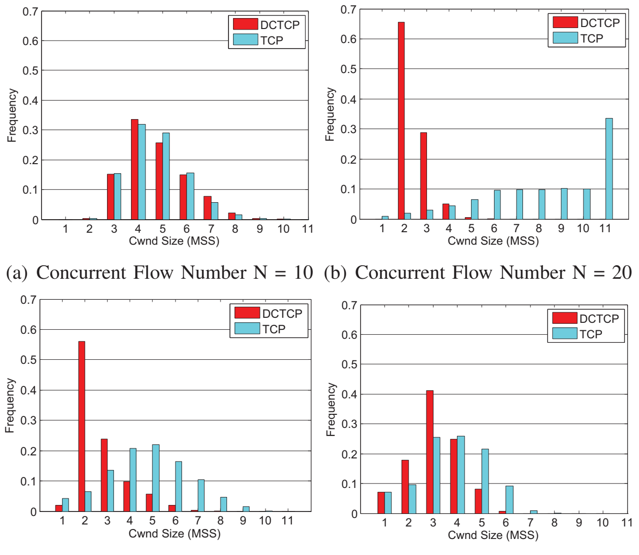
A. Throughput collapse for DCTCP under the high fan-in traffic pattern

We redo the basic incast experiment in [1] on our testbed. The aggregate server requests 1MB/N bytes from N concurrent senders through ECN-enabled switches with the static 128KB buffer per port and threshold K is set to 32KB. Each receiver responds immediately with the requested data. The aggregator will wait until it receives all responses, and then issues another similar requests. The experiment is repeated 1000 times with N varying from 1 to 100.

Figure 1 shows the throughput of DCTCP and TCP versus the number of concurrent flows. The results are the same with those in [1]: DCTCP begins to suffer from throughput collapse when the number of concurrent flows exceeds 35. TCP undergoes the collapse just exceeding 10. We trace all the congestion window (`cwnd`) size evolution and the `ECE` flag bit in TCPs headers of all concurrent flows, find some particular clues when DCTCP suffers from the performance impairment.

B. DCTCP is incapable of further reducing the sending rate when `cwnd` reaches the minimum size

In the above experiment, we make a snapshot of `cwnd` and `ECE` when N equals to 10, 20, 40 and 60 respectively. Under



(a) Concurrent Flow Number N = 10 (b) Concurrent Flow Number N = 20

(c) Concurrent Flow Number N = 40 (d) Concurrent Flow Number N = 60

Fig. 2. The frequency distribution of `cwnd` sizes for DCTCP and TCP when the number of concurrent flow is 10, 20, 40, 60

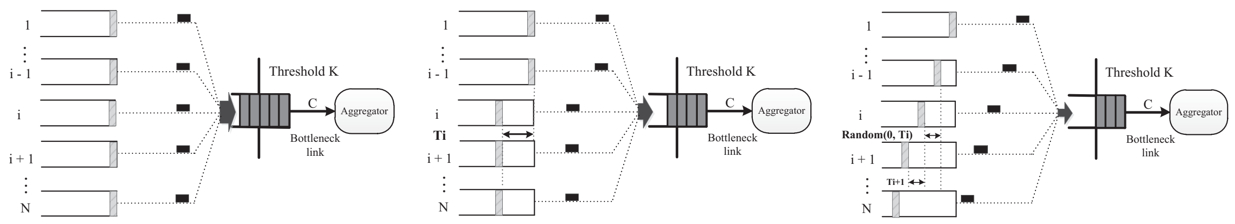
the high fan-in traffic pattern, DCTCP is usually unable to further decrease its sending rate when `cwnd` reaches the lower bound while there still receives the congestion information generated by switches.

Naturely, the window-based congestion control mechanism has the range and granularity limitation for window size adjustment. As Equation (1) and 2 show, the window size has its lower bound value, 2MSS.[2] In the experiment, we observed a higher frequency for DCTCP's `cwnd` hitting the lower bound as the number of concurrent flows increases. Figure 2 shows the frequency distribution of `cwnd` sizes. When $N = 10$, the `cwnd` size for DCTCP and TCP mainly distribute from 3 to 8MSS, and $< 1\%$ distribute in 2, 9 and 10MSS. However, when N = 20, 40 and 60 respectively, $+60\%$ of DCTCP's `cwnd` size ranges from 1MSS to 2MSS, among which great majority of `cwnd` sizes equals to 2MSS (The remaining `cwnd` = 1MSS indicates timeouts). For TCP, similar phenomenon happen as N increases. But compared with DCTCP, TCP is relatively less sensitive to congestions and lags in control from the distribution of TCP's and DCTCP's `cwnd` in Figure 2.

Significantly, with the number of flows increasing, the ratio of timeouts (namely, `cwnd=1`) increases while the ratio of `cwnd=2` decreases as showed from Figure 2(b), 2(c) to 2(d). This indicates that the higher fan-in traffic shares the same bottleneck link, the more risk DCTCP takes to suffer from timeouts.

In order to further confirm DCTCP's limitation, we trace the `cwnd` and `ECE` flag bit variation of one flow randomly selected. If `cwnd=2, ECE=1` before sending packets, it evidently indicates that DCTCP is unable to decrease the sending rate any more although it is required by `ECE`. As

[2]For most Linux kernel distributions, the normal minimum value for `cwnd` is 2MSS except the 1MSS for the flow initialization and timeout occurrence.



(a) Synchronized N DCTCP flows with the minimum cwnd make buffer overflow

(b) Regulating flows' sending time interval when cwnd reaches the minimum

(c) Randomizing flows' sending time interval when cwnd reaches the minimum

Fig. 3. Basic idea for DCTCP$^+$

TABLE I
PERCENTAGE FOR ONE CONCURRENT FLOW ENCOUNTERING THE CASE OF (1) `cwnd=2, ECE=1`, (2) `Timeout` AMONG ALL TRANSMISSIONS, (3) FLOSS-TO AMONG ALL TIMEOUTS, (4) LACK-TO AMONG TIMEOUTS

Flow Number	cwnd=2, ECE=1	Timeout		FLoss-TO	LAck-TO
	DCTCP	DCTCP	TCP	DCTCP	DCTCP
N=20	58.30%	0	0.95%	0	0
N=40	50.16%	1.9%	4.23%	35.23%	64.77%
N=60	10.41%	7.07%	7.18%	76.33%	23.67%

Table I shows, among all the tranmissions, this incapable case accounts for 58.30% for $N = 20$, 50.16% for $N = 40$ and 10.41% for $N = 60$ in of DCTCP. This not only reflects the limited congestion control ability for DCTCP based on the window size under the high fan-in traffic, but also brings the threat for timeouts as we will elaborate next.

C. *Bursts from synchronized flows with small* `cwnd` *cause fatal packet loss leading to severe timeouts*

We collect the frequency of the times of timeouts among all times of transmissions for one flow randomly selected. Table I shows the increasing percentage of timeouts for both DCTCP and TCP as the synchronized flow number increases.

Carefully examining the states of protocol stack when timeout occurs, we find two main categories of timeouts. One is FLoss-TO [12], which is induced by the full window loss. The other is LAck-TO [12], which is caused by the insufficient packets for data-driven recovery, e.g. less than triple duplicate ACKs received by the sender. These two kinds of timeouts account for all timeout cases in experiments as Table I shows. Moreover, the more synchronized flows there are, the more FLoss-TOs occur.

These two timeouts are both resulted from the fatal packet losses that synchronized N concurrent flows' packets overflow Pipeline Capacity. Let's make a simple calculation. Suppose $N = 40$, Pipeline Capacity $C \times D + B$ is $100Gbps \times 100us + 128KB = 140.5KB$. If flow i's window size $w(i,t) = 2$MSS, $\sum_{i=1}^{N} w(i,t) = 2 \times 1.5 \times 40 = 120KB$ approaches Pipeline Capacity. If $w(i,t) = 3$MSS, $\sum_{i=1}^{N} w(i,t) = 180KB$ exceeds Pipeline Capacity. When $N = 60$, even if $w(i,t) = 2$MSS, $\sum_{i=1}^{N} w(i,t) = 180KB$ also exceeds Pipeline Capacity. More importantly, any packet losses under this condition easily lead to FLoss-TO and LAck-TO.

From the analysis above, we find the two key factors responsible for DCTCP's performance impairment under the high fan-in traffic. The first is the inability to further decrease the sending rate when `cwnd` hits the lower bound in the face of congestion. The second is the synchronization of the concurrent flows' sending makes the fan-in burst exceed the small pipeline capacity of data center networks, then leading to packet losses, which causes awful FLoss-TOs and LAck-TOs directly.

V. DCTCP$^+$ DESIGN AND IMPLEMENTATION

Guided by the above findings, we design DCTCP$^+$ protocol to support massive concurrent flows. Firstly, we introduce the basic idea of design DCTCP$^+$.

A. *Basic idea for DCTCP$^+$*

Our starting point is whether we could introduce some additional mechanisms into DCTCP$^+$ to eliminate or greatly alleviate the performance impairments caused by two main factors. In order to achieve this goal, we consider to solve the problem in two stages aiming at the key factors respectively.

The first step is to enhance DCTCP so that it can decrease the sending rate in case of the high fan-in congestion, when its `cwnd` reaches to the minimum value and still receives congestion notification. Straightforwardly, we can introduce the enhancement mechanism to enlarge the sending time interval. Equivalently, it can slow down the sending rate required by ECN feedback. In other words, the sender will wait for a `slow_time` to inject the next packet into networks instead of immediate transmission.

Figure 3(a) shows N concurrent DCTCP flows overwhelm the switch shallow buffer when their `cwnd`s reach the minimum value. Figure 3(b) illustrates our basic idea. When the sender is aware that it reaches the minimum window size driven by the regulating law of DCTCP, it begins to delay packet transmission to regulate the sending time interval. We believe that it deserves to pause hundreds to thousands of microseconds to send the next packet. Although it increases the completion time a little, it is expected to be a worthy trade-off to avoid the severe timeouts, which otherwise will introduce hundreds of milliseconds.

The first step would help decrease the sending rate. However, it does not fix the second problem entirely. If the massive concurrent flows send packets simultaneously, the

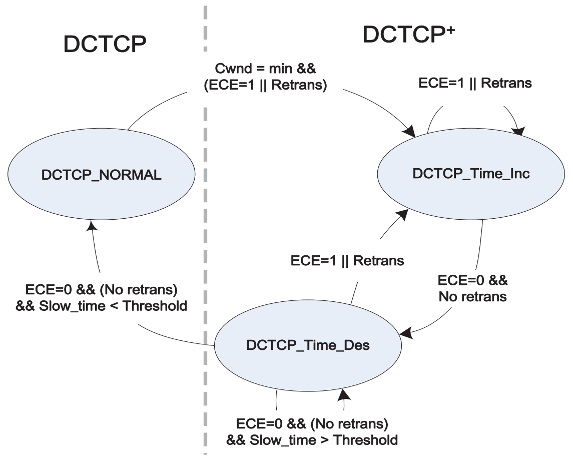


Fig. 4. State Machine Diagram of DCTCP$^+$

synchronized packets would produce micro burst at the bottleneck link. Therefore, for the problem shown in in Figure 3(b) that bursts from synchronized flows with small `cwnd` cause fatal packet loss and lead to severe timeouts, we try to randomize the sending time of concurrent flows, avoiding the synchronized bursts as shown in Figure 3(c). During the experiments in Section VI, we will find this is quite necessary if DCTCP$^+$ needs to support more concurrent flows. If we do not desynchronize the sending time interval randomly, DCTCP$^+$ could only support about 100 concurrent flows well on our testbed and then performs poorly like DCTCP for more concurrent flows.

The entire DCTCP$^+$ design is centered around two ideas: to further decrease the sending rate to avoid the fan-in congestion, and to desynchronize the concurrent flow to alleviate fan-in bursts.

B. State transition between DCTCP$^+$ and DCTCP

We define three important states during designing of DCTCP$^+$. In the different states, the sender will be delayed for different `slow_time` to send the next packet. These states are:

State-I `DCTCP_NORMAL`: represents DCTCP works normally. In this state, the sender's `cwnd` is equal to or greater than its lower bound.

State-II `DCTCP_Time_Inc`: represents that `cwnd` has diminished to the minimum value, and meanwhile the sender is notified to further decrease the sending rate, induced by the ACKs with ECN marked or retransmission after the timeout. In this state, the packet transmission is delayed by `slow_time` to decrease the sending rate on average.

State-III `DCTCP_Time_Des`: represents a recovering state. If the sender is in `DCTCP_Time_Inc` state and receives no more congestion information, the state machine will transit into `DCTCP_Time_Des` state. After the further regulation of `DCTCP_Time_Des`, if no congestion feedback is received, the sender will return to DCTCP from DCTCP$^+$.

The state transitions and the corresponding conditions are summarized in Figure 4. The transition condition `ECE=1` means ACK from the receiver is marked with the ECN tag. The condition `retrans` and `no retrans` stands for whether the retransmission action indicating the occurrence of timeouts is triggered or not. `Threshold` is a time threshold to guarantee the relatively smooth regulation of the sending rate from `DCTCP_Time_Des` to `DCTCP_NORMAL`.

C. Sending Time Interval Regulation

The sending time interval regulation is critical for each concurrent flow to quickly converge to proper `slow_time`. The regulation law should consider not only to decrease the whole sending rate of all concurrent flows, but also to avoid bursts from the synchronized senders. In DCTCP$^+$, we use a heuristic algorithm to regulate the sending time interval.

The critical variable `slow_time` is updated following the additive-increase/multiplicative-decrease (AIMD) rule in DCTCP$^+$. As Algorithm 1 shows, from state `DCTCP_NORMAL` to `DCTCP_Time_Inc`, the `slow_time` is increased by one `backoff_time_unit` from zero, e.g. 100 us is the default for `backoff_time_unit`. Every time `DCTCP_Time_Inc` transits to `DCTCP_Time_Inc`, the `slow_time` is increased by one time unit. The sending rate, inversely proportional to the sending time interval, is reduced accordingly. This additive-increase method for `slow_time` intends to find a conservative rate for senders based on the congestion information.

Every time `DCTCP_Time_Inc` state transfers into `DCTCP_Time_Des`, the `slow_time` is divided by a factor, e.g. 2, 4. Until `slow_time` is less than `threshold_T`, the state will transit into `DCTCP_NORMAL`. The corresponding sending rate is multiplied to rapidly recover to the normal state. To avoid synchronized bursts, we randomize the sending time by making time unit `backoff_time_unit` evenly distributed for `slow_time` as shown in Algorithm 1.

What we have to declare is the heuristic algorithm is not the optimal `slow_time` regulation. Here, we just provide a plain heuristic algorithm for the regulation of `slow_time` which does work in experiments. Further investigation would focus on the fine regulation law for `slow_time`.

D. Implementation

We implement the DCTCP$^+$ on Linux-Kernel-2.6.38.3 through modifying the available DCTCP source code found in [18]. The implementation of DCTCP$^+$ requires less than 100 lines of code change to DCTCP. We use `ndctcp_status_evolution()` function to maintain the state machine shown in Figure 4, and change the sending time interval `slow_time` according to Algorithm 1. Once the sender receives one ACK, `ndctcp_status_evolution()` will be invoked to check whether state transition update is required or not. Before packets entering the `tcp_transmit_skb()` function, the

Algorithm 1 Sending time interval regulation

Symbols Descriptions:
backoff_time_unit: basic time unit for the backoff;
divisor_factor: divisor to decrease the long time interval;
isTo<NextState>: procedure to see if the current protocol states meet the transition conditions in Fig 4;

```
 1: procedure STATUSES_EVOLUTION($current\_state$)
 2:     switch $current\_state$ do
 3:         case $DCTCP\_NORMAL$
 4:             if $isToDCTCP\_Time\_Inc$ then
 5:                 $current\_state = DCTCP\_Time\_Inc$
 6:                 $slow\_time = random(backoff\_time\_unit)$
 7:                 $Break$
 8:             end if
 9:         case $DCTCP\_Time\_Inc$
10:             if $isToDCTCP\_Time\_Inc$ then
11:                 $slow\_time+ = random(backoff\_time\_unit)$
12:                 $Break$
13:             end if
14:             if $isToDCTCP\_Time\_Des$ then
15:                 $current\_state = DCTCP\_Time\_Inc$
16:                 $slow\_time/ = divisor\_factor$
17:                 $Break$
18:             end if
19:         case $DCTCP\_Time\_Des$
20:             if $isToDCTCP\_Time\_Inc$ then
21:                 $current\_state = DCTCP\_Time\_Des$
22:                 $slow\_time+ = random(backoff\_time\_unit)$
23:                 $Break$
24:             else if $slow\_time > threshold\_T$ then
25:                 $slow\_time/ = divisor\_factor$
26:             else
27:                 $current\_state = DCTCP\_NORMAL$
28:             end if
29: end procedure
```

current state is checked, and the corresponding `slow_time` is delayed through the callback of `tcp_transmit_skb()` which is invoked by an high-resolution timer (`hrtimer`) [19] to regulate the sending time of flows deliberately.

Here we introduce some guidelines towards parameter setting in DCTCP$^+$. First, we choose to use the baseline RTT as the backoff time unit. It is advised neither to use the large time unit since it could reduce the sending rate too much and lead to bandwidth wastage, nor to use the small time unit because it could not help relieve the severe congestion caused by high fan-in traffic. Second, we use the divisor factor 2 to reduce `slow_time` to recover from the congestion. Similarly, the divisor factor is suggested neither to be too big for the premature recovery from the congestion state to the normal, nor too conservative for retarding sending rate regulation.

VI. Performance Evaluation

In this section, we verify our design idea and evaluate the performance among DCTCP$^+$, DCTCP and TCP using some real benchmarks on our testbed, including the basic incast experiment without background traffic, the incast experiment with background traffic and the benchmark traffic pattern from the production clusters found in [1].

A. Evaluation Methodology

We deployed the full implementation of DCTCP$^+$ on all servers in experiments. The CPU, memory, or hard disk of the servers were never the bottleneck in any experiments. The

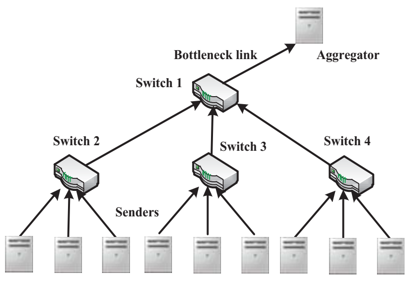


Fig. 5. Topology of basic incast scenario

topology is shown in Figure 5 which is a canonical tree-based 2-Tier topology. Each switch has a static 128KB shared buffer in each port. The threshold K of DCTCP is set to 32KB as recommended in [1]. We focus on the performance comparison among DCTCP$^+$, DCTCP, and TCP New Reno, and validating the effectiveness of enhancement mechanisms in DCTCP$^+$.

B. Basic Incast without Background Traffic

Incast traffic [16][20][21] usually involves large number of concurrent flows, characterized by the high fan-in congestion. We examine the performance of DCTCP$^+$, DCTCP and TCP under the incast experiment first.

In the experiment, the aggregator requests 1 MB/N bytes from N workers. The concurrent flows are established by multiple threads between the aggregator and the nine servers in a serially round-robin way. The worker responds immediately with the requested data. The aggregator will wait until receiving all the responses and then issue another similar requests to workers. The experiment is repeated 1000 times for every N concurrent flows and N varies from 1 to 200.

In order to validate two enhancement mechanisms against two key problems that DCTCP fails to handle the high fan-in traffic, we will verify the performance of DCTCP$^+$[3] one by one. First, to examine whether DCTCP$^+$ further decreases the sending rate when its `cwnd` reaches the minimum value, we just regulate the sending time interval `slow_time` in DCTCP$^+$ without scattering it to avoid the synchronized burst, namely only one mechanism is enabled. As shown in Figure 6, the problem of DCTCP$^+$ is overcomed partially. Although DCTCP$^+$ maintains a higher throughput without timeouts, synchronized bursts couldn't be avoided when there are more concurrent flows. The throughput of DCTCP$^+$ sharply declines when the concurrent flow number exceeds 100. This tells that it's not enough just to decrease the sending rate, because the window-based flow control is not fine-grained for the sending rate and the aggregation traffic from many simultaneous senders produces bursts on the bottleneck link.

Then, we implement the completed DCTCP$^+$ considering the synchronization mechanism flows by randomizing the

[3]We change the lower bound of `cwnd` to 1MSS for DCTCP$^+$ for the smoother change of the sending rate. We also do this for DCTCP for comparison. Actually it doesn't improve DCTCP's performance in the high fan-in traffic according to the analysis in Section IV.

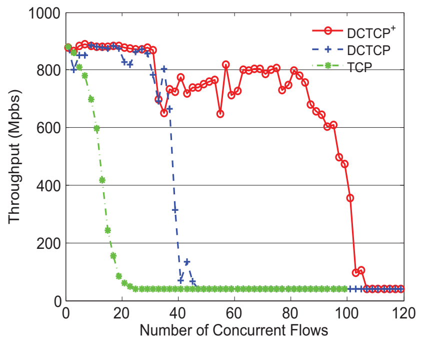


Fig. 6. Partially implemented DCTCP$^+$ in the incast experiment

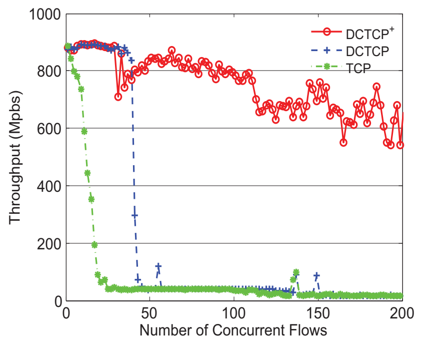


Fig. 7. Fully implemented DCTCP$^+$ in the incast experiment

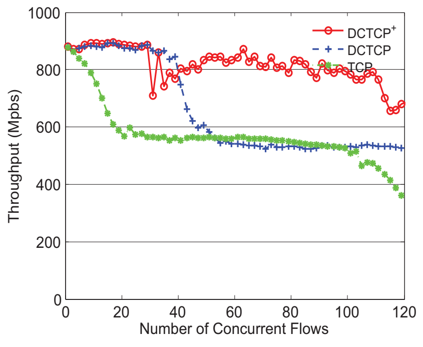
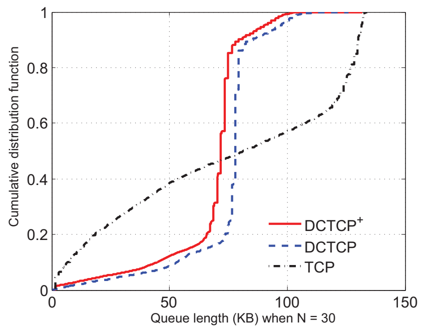
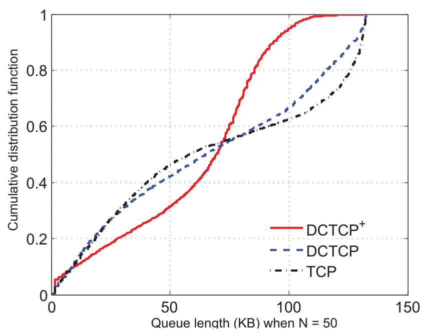


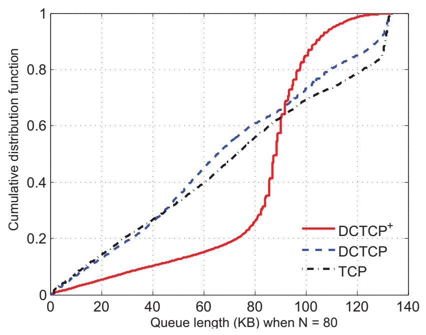
Fig. 8. DCTCP$^+$ compared with the 10ms RTO DCTCP and TCP in the incast experiment



(a) CDF of queue length when $N = 30$



(b) CDF of queue length when $N = 50$



(c) CDF of queue length when $N = 80$

Fig. 9. CDF of queue length variation on Switch 1 when concurrent flow number $N = 30, 50, 80$

sending time. Figure 7 confirms that DCTCP$^+$ can support more than 200 concurrent flows nicely. In normal conditions, DCTCP$^+$ performs as well as DCTCP when the concurrent flow number is within 40. Under the high fan-in traffic, DCTCP$^+$ keeps a high throughput, which fluctuates between 600 and 900Mbps even when the concurrent flows exceeds 200. This provides the advantage for DCTCP$^+$ in aspect of FCT compared with DCTCP and TCP. DCTCP$^+$ holds the short FCT ranging from 8ms to 17ms as the concurrent flow number increases to 200. Both DCTCP and TCP all suffer from severe timeouts resulting in FCT being more than 200ms.

Besides, due to the relatively large default value of RTO, i.e. 200ms, which penalizes the performance greatly as proved in [16], we modify the RTO time to 10ms for DCTCP and TCP for a fair comparison in the above experiment. Figure 8 shows the throughput for DCTCP$^+$, DCTCP and TCP respectively. The default RTO setting is not modified for DCTCP$^+$, while it outperforms DCTCP and TCP without timeouts. Although we could see the improvement in throughput for DCTCP and TCP by quick retransmissions after timeouts, we do not recommend to arbitrarily decrease RTO since it will bring spurious timeouts and other problems [22].

Two details deserve attentions as shown in Figure 7. First, even when we have decreased the minimum value of `cwnd` to 1MSS, DCTCP still suffers from the incast impairment. Actually, this result accords with our analysis in Section IV, decreasing the minimum value of `cwnd` won't help DCTCP improve performance. Second, we observe that there is an obvious decline in throughput for both DCTCP$^+$ and DCTCP when the concurrent flow number is between 30 and 40. In fact, DCTCP$^+$ takes actions to decrease the sending rate during this period. But DCTCP has researched the critical point of congestion adjustment at that moment.

In experiments, we collect the instant queue length every 100us on *Switch 1*. Figure 9 shows the cumulative distribution function (CDF) of the switch queue length for DCTCP$^+$, DCTCP and TCP. When $N = 30$, DCTCP$^+$ begins to show a shorter queue length than DCTCP. We could see the evident difference in queue length between DCTCP$^+$ and DCTCP from Figure 9(b) to 9(c). DCTCP$^+$ keeps a much shorter and stabler queue length than DCTCP and TCP. This proves DCTCP$^+$ does work in the incast experiment.

C. Incast Congestion with Background Traffic

We do not consider the impact of the background traffic in the above experiment. To evaluate the performance of DCTCP$^+$ in the incast experiment mixing with background traffic, we start two persistent flows to consume the shared buffer as Figure 10 shows.

Figure 11 and 12 show the good throughput and FCT versus the number of concurrent flows respectively. Although background traffic shares the same bottleneck link in Figure

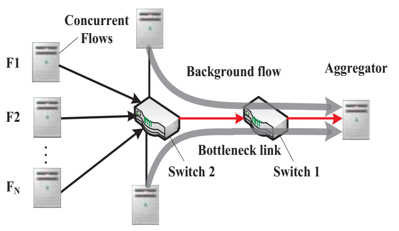


Fig. 10. Incast scenario with background traffic

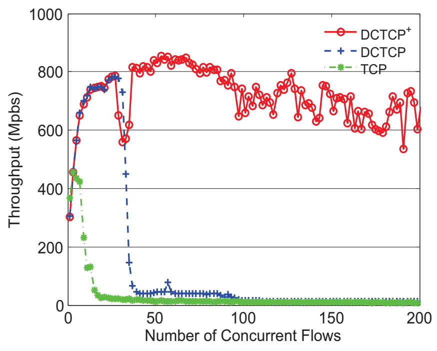


Fig. 11. Good throughput (Mbps) for the incast scenario with background traffic

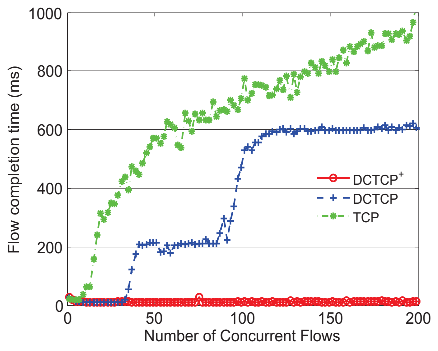
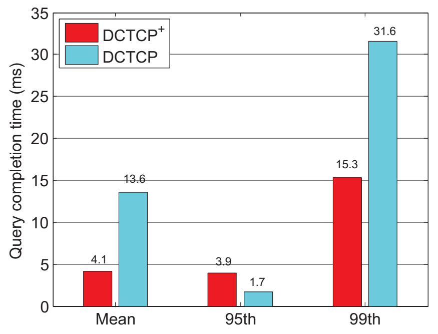
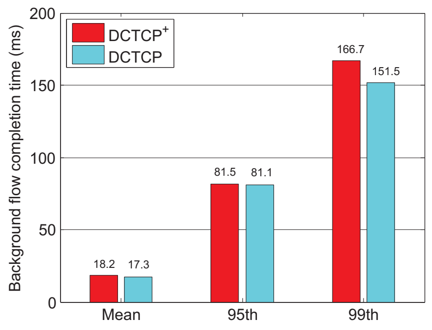


Fig. 12. Flow completion time (ms) with background traffic



(a) Query completion time



(b) Background flow completion time

Fig. 13. The FCT for DCTCP$^+$ and DCTCP under the benchmark traffic

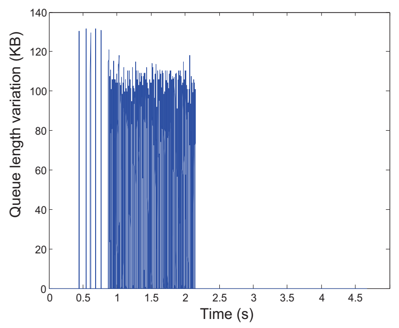


Fig. 14. *Switch 1* queue length every 100us for DCTCP$^+$ when $N = 50$

10, we can see that DCTCP$^+$ still outperforms DCTCP and TCP and keeps almost as good throughput without long background flows. The apparent gap in FCT can be seen from Figure 12. Although DCTCP$^+$ experienced hundreds to thousands milliseconds delay, its FCT is much shorter than DCTCP and TCP. This is what we mean slowing little quickens more.

Besides, we collect the average throughput of two DCTCP$^+$ long flows every time transmitting 1GB data. The average throughput for both long flows is around 400Mbps. DCTCP$^+$ provides better performance isolation between short and long flows than DCTCP.

D. *Benchmark Traffic*

We generated the realistic traffic including query, short messages and background flows, based on statistics from the production cluster [1]. Query traffic sent to all servers follows the inter-arrival time distribution in [1] with 2KB responses. Short messages and background traffic are produced according to the flow size versus the inter-arrival time distribution from the measurement result of the production cluster in [1]. We carry out the experiment for DCTCP$^+$ and DCTCP with both RTO_{min} set to be 10ms. Threshold K for the switch buffer is set to be 32KB and the static shared buffer size is set to be 128KB for each port of the switch. The traffic generated consists of 7,000 queries and 7,000 background flows.

Figure 13(a) shows the statistical result of the queries' FCT for DCTCP$^+$ and DCTCP. DCTCP$^+$ performs better than DCTCP on the whole. For the mean value, the queries' FCT for DCTCP$^+$ is 4.1ms with fewer timeouts. However, the mean queries' FCT for DCTCP is 13.6ms, indicating timeout happens at least once in experiments on average after changing default RTO_{min} 10ms. For the 95th percentile of the queries' FCT for DCTCP$^+$ and DCTCP, it confirms the fact that adjusting the sending time interval will increase the flow completion time if no timeout happening. However, for the 99th percentile of FCT, we can see the advantage of DCTCP$^+$ over DCTCP. Flows at the tail-end of the FCT distribution benefit significantly from DCTCP$^+$ with 16.3ms gain.

Figure 13(b) shows the FCT for the background traffic. Although there is less than 1ms difference between DCTCP$^+$ and DCTCP for the mean and 95th percentile FCT and 15.2ms for the 99th percentile FCT, there is no much difference on the whole. These tiny gaps can be ascribed to the hundreds to thousands of microseconds delay introduced by DCTCP$^+$. Nevertheless, it doesn't bring much negative influence for the background flow.

Under the benchmark traffic, DCTCP$^+$ could achieve shorter FCT for the query traffic, avoiding timeouts which put off the entire DCTCP's FCT. And the impact from DCTCP$^+$'s regulation is minor for the background traffic.

VII. Discussions

In this section, we discuss three aspects related to the further extension of DCTCP$^+$.

The first is about finer backoff time design for DCTCP$^+$. We could clearly observe the small fluctuation on throughput for DCTCP$^+$ in Figure 7 and 11. As the number of concurrent flows increases, the fluctuation is more drastic. This problem comes from the fact the regulation of DCTCP$^+$ is not much smooth. The more concurrent flows swarm into the bottleneck link, the more precise control on sending rate is needed. In this work, we just provide the algorithm proving that the design works and performs well. Further finer algorithm about regulating the sending time interval is appreciated.

The second is the extension of enhancement mechanism with other transmission control protocols. Although we design DCTCP$^+$ based on DCTCP, the fact is the idea of enhancement mechanism could be coalesced with other data center protocols, for example, D^{2}TCP [2], the HULL architecture [3] and TCP protocol. Protocols with integration of the enhancement mechanism are expected to better handle the high fan-in traffic in data center network.

The third is about the convergence speed of DCTCP$^+$. Frankly speaking, DCTCP$^+$ is unable to work in the first RTT round in the high fan-in traffic, because there is no congestion information back from the receivers. DCTCP$^+$ needs several cycles of RTTs to enter the enhancement mechanism. Thus, it's easy to cause timeouts at the first rounds. Figure 14 shows when the aggregator requested 4MB data for every one of the 50 concurrent flows, the buffer overflowed in the initial five rounds. However, fortunately, the statistical flow size for traffic in data center is tens to hundreds of megabytes [1][7]. It is enough for DCTCP$^+$ to converge to the stable state. Even for the query responded with several kilobytes, the situation is not so bad coupled with the background traffic. For the initial timeout problem, we think it is hard to avoid just through the protocol's congestion control. It is better to rely on some mechanisms similar to Connection Admission Control for the historical ATM networks [23].

VIII. Related Work

TCP Pacing: TCP pacing [25-27] is suggested for alleviating the burst because of the ACK compression. The adjustment of DCTCP$^+$ to regulate the send time interval is similar to TCP Pacing. They are all aimed at controlling the send rate of the senders. However, the primary purpose for pacing is to smooth the change of the send rate to avoid the bursts of traffic injected into the network causing network congestion. Pacing could increase or decrease the send rate. But the adjustment in DCTCP$^+$ aims to slow down the send rate, to avoid the overflow of the aggregator layer's switch buffer due to large number of concurrent flows, and to avoid packet drops and timeouts to finish responding the entire requests before the deadline. What's more, pacing is not unanimous as Aggarwal et al. [24] show that the paced flows are negatively impacted when competing with the non-paced flows since they are deliberately delayed. For DCTCP$^+$, this problem does not exist. Firstly, DCTCP$^+$ is based on DCTCP and works well like DCTCP in the normal network environment. DCTCP$^+$ regulates the send time interval only in the special case when large number of concurrent flows swarms into the network. Secondly, data center owns the homogeneity trait. All the servers with DCTCP$^+$ receive the request from the aggregator and act almost in concert with each other without competition.

Transport Protocols in Datacenters: Standard TCP has aroused many problems in data centers [16][20]. The large number of concurrent flows is common with the partition aggregate application pattern which could cause a severe decline in network throughput. What's more, the throughput-sensitive long flows combined with the delay sensitive short flows coexist in the data center network. Long flows hurt the short flows since long flows drive queues to loss under the control of TCP protocol.

Motivated by these problems, recent proposals focus on the design of novel congestion control protocols to address the problems [25]. DCTCP employs the Explicit Congestion Notification mechanism to control the queue length within a small value in order to reduce the latency for short flows, maintain high throughput for long flows, and provide certain burst tolerance ability. Since DCTCP only decreases latency for short flows without providing differentiated services for flows with different delay requirement, D^{2}TCP extends the window evolution function of DCTCP. The flows with smaller remaining delay will obtain higher rates. In D3 [26], end hosts compute the desired rate for each latency-sensitive flow based on the flow size and deadline, convey the computed rates to the switch. The switches then allocate the corresponding link rates to each flow based on the collected rate value. These protocols attempt to control the flow rate according to the delay information, while some other protocols like PDQ [27] reduce the latency for delay-sensitive flows through priority scheduling at the switches. These transport protocols in data center are all aimed at: (1) low latency for the latency-sentitive flows; (2) high burst tolerance; (3) high utilization for long flows.

IX. Conclusion

In this paper, we propose DCTCP$^+$ to enhance DCTCP under the high fan-in traffic pattern. Our work is motivated by the fact DCTCP fails to handle the congestion resulting from high fan-in traffic. Although DCTCP has been recognized for its effectiveness in algorithm and simplicity in deployment, this shortcoming will make DCTCP suffer from severe performance impairment in some real production clusters, especially those employing divide-and-conquer computing paradigm.

We trace and analyse DCTCP's dynamic behavior at senders in real experiments, reveal two radical reasons for DCTCP's failure under the high fan-in concurrent traffic: (1) DCTCP is unable to further decrease the sending rate when `cwnd` has been decreased to the minimum value ; (2) The synchronized bursts from the concurrent senders make FLoss-TOs and LAck-TOs. We enhance DCTCP through regulating sending time interval to decrease the sending rate and desynchronizing sending time to avoid the synchronized bursts.

DCTCP$^+$ shows significant improvement in the high fan-in traffic. On the one side, DCTCP$^+$ inherits good performance of DCTCP for low latency, high burst tolerance, high utilization for long flows. On the other side, DCTCP$^+$ keeps the lower switch buffer occupancy, higher throughput, and shorter FCT when the number of concurrent flows increases to hundreds. It performs better for the short concurrent queries in the benchmark traffic, and keeps the same good performance for the background traffic.

X. Acknowledgment

The authors gratefully acknowledge the anonymous reviewers for their constructive comments. This work is supported in part by National Basic Research Program of China (973 Program) under Grant No. 2012CB315803, and National Natural Science Foundation of China (NSFC) under Grant No. 61225011

References

[1] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data center tcp (dctcp)," ser. SIGCOMM '10. New York, NY, USA: ACM, 2010, pp. 63–74.

[2] B. Vamanan, J. Hasan, and T. Vijaykumar, "Deadline-aware datacenter tcp (d2tcp)," *SIGCOMM Comput. Commun. Rev.*, vol. 42, no. 4, pp. 115–126, Aug. 2012.

[3] 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," ser. NSDI'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 19–19.

[4] R. S. M. Tüxen and G. V. Neville-Neil, "An investigation into data center congestion control with ecn," http://www.bsdcan.org/2011/schedule/events/242.en.html, 2011.

[5] T. Maurer, "Windows server 2012: Datacenter tcp (dctcp)," http://www.thomasmaurer.ch/2012/07/windows-server-2012-datacenter-tcp-dctcp/, Jul. 2012.

[6] Yahoo!, ""m45 supercomputing project"," http://research.yahoo.com/node/1884, 2009.

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

[8] V. Jalaparti, P. Bodik, S. Kandula, I. Menache, M. Rybalkin, and C. Yan, "Speeding up distributed request-response workflows," *SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 4, pp. 219–230, Aug. 2013.

[9] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," ser. IMC '10, New York, NY, USA, 2010, pp. 267–280.

[10] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken, "The nature of data center traffic: measurements & analysis," ser. IMC '09. New York, NY, USA: ACM, 2009, pp. 202–208.

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

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

[13] S. Kavulya, J. Tan, R. Gandhi, and P. Narasimhan, "An analysis of traces from a production mapreduce cluster," in *Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on*, 2010, pp. 94–103.

[14] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008.

[15] A. P. Vijay Vasudevan, E. K. Hiral Shah, G. R. G. David G. Andersen, and G. A. Gibson, "A (in)cast of thousands: Scaling datacenter tcp to kiloservers and gigabits," Technical Report CMUPDL-09-101, Feb. 2009.

[16] 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," *SIGCOMM Comput. Commun. Rev.*, vol. 39, no. 4, 2009.

[17] http://sourceware.org/systemtap/kprobes/.

[18] "Data center tcp," http://www.http://simula.stanford.edu/~alizade/Site/DCTCP.html, Jul. 2012.

[19] http://www.ibm.com/developerworks/library/l-timers-list/.

[20] Y. Chen, R. Griffith, J. Liu, R. H. Katz, and A. D. Joseph, "Understanding tcp incast throughput collapse in datacenter networks," in *Proceedings of the 1st ACM workshop on Research on enterprise networking*, ser. WREN '09, New York, NY, USA, 2009, pp. 73–82.

[21] A. Phanishayee, E. Krevat, V. Vasudevan, D. G. Andersen, G. R. Ganger, G. A. Gibson, and S. Seshan, "Measurement and analysis of tcp throughput collapse in cluster-based storage systems," ser. FAST'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 12:1–12:14.

[22] L. Cheng, C.-L. Wang, and F. Lau, "Pvtcp: Towards practical and effective congestion control in virtualized datacenters," in *Network Protocols (ICNP), 2013 21st IEEE International Conference on*, Oct 2013, pp. 1–10.

[23] H. G. Perros and K. M. Elsayed, "Call admission control schemes: a review," *Comm. Mag.*, vol. 34, no. 11, pp. 82–91, Nov. 1996.

[24] A. Aggarwal, S. Savage, and T. Anderson, "Understanding the performance of tcp pacing," in *INFOCOM 2000. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, vol. 3, 2000, pp. 1157–1165 vol.3.

[25] J. Zhang, F. Ren, and C. Lin, "Survey on transport control in data center networks," *Network, IEEE*, vol. 27, no. 4, pp. 22–26, 2013.

[26] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowtron, "Better never than late: meeting deadlines in datacenter networks," ser. SIGCOMM '11. New York, NY, USA: ACM, 2011, pp. 50–61.

[27] C.-Y. Hong, M. Caesar, and P. B. Godfrey, "Finishing flows quickly with preemptive scheduling," ser. SIGCOMM '12. New York, NY, USA: ACM, 2012, pp. 127–138.