FaST: Fine-grained and Scalable TCP for Cloud Data Center Networks

Jaehyun Hwang[1] and Joon Yoo[2]
[1] Bell Labs, Alcatel-Lucent
Seoul, Republic of Korea
[e-mail: jh.hwang@alcatel-lucent.com]
[2] Department of Software Design & Management, Gachon University
Seongnam, Republic of Korea
[e-mail: joon.yoo@gachon.ac.kr]
*Corresponding author: Joon Yoo

Received September 13, 2013; revised January 24, 2014; accepted February 15, 2013; published March 31, 2014

Abstract

With the increasing usage of cloud applications such as MapReduce and social networking, the amount of data traffic in data center networks continues to grow. Moreover, these applications follow the *incast* traffic pattern, where a large burst of traffic sent by a number of senders, accumulates simultaneously at the shallow-buffered data center switches. This causes severe packet losses. The currently deployed TCP is custom-tailored for the wide-area Internet. This causes cloud applications to suffer long completion times towing to the packet losses, and hence, results in a poor quality of service. An Explicit Congestion Notification (ECN)-based approach is an attractive solution that conservatively adjusts to the network congestion in advance. This legacy approach, however, lacks scalability in terms of the number of flows. In this paper, we reveal the primary cause of the scalability issue through analysis, and propose a new congestion-control algorithm called FaST. FaST employs a novel, virtual congestion window to conduct fine-grained congestion control that results in improved scalability. Furthermore, FaST is easy to deploy since it requires only a few software modifications at the server-side. Through ns-3 simulations, we show that FaST improves the scalability of data center networks compared with the existing approaches.

Keywords: Scalable congestion control, cloud data center networks, virtual congestion window

This research was supported by the Seoul R&BD Program (WR080951) funded by the Seoul Metropolitan Government and in part by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT & Future Planning (NRF-2013R1A1A1006823).

http://dx.doi.org/10.3837/tiis.2014.03.003

1. Introduction

Cloud data center applications such as MapReduce [1], social networking [2], and recommendation systems [3] demand a severe latency requirement because application latency affects user quality-of-service, and hence, operator revenue. These applications generally employ the Partition/Aggregate design pattern. Applications divide and distribute the jobs to multiple servers, and then, one receiver (called the aggregator) simultaneously aggregates the response data from the multiple servers (called the workers). However, the currently deployed legacy TCP shows poor performance for cloud data center applications because of the following reasons.

First, the Partition/Aggregate traffic pattern causes *incast* congestion, where the main bottleneck point is at the Top-of-the-Rack (ToR) switches. Today's data center ToR switches generally use a small buffer memory to reduce cost. This results in frequent buffer overflows, and hence, packet losses. Second, scalability is a critical issue since recent reports on data center networks show that the average number of flows in a ToR switch exceeds a few tens and these are consistently growing [7][17]. Third, TCP senders frequently experience timeouts since they rely on retransmission timeouts (RTO) to detect packet losses. However, the RTO is not a good indication because the minimum RTO (RTO_{min}) is normally in the order of milliseconds (that is, 200-300 ms) while the Round Trip Times (RTTs) are in the order of microseconds (that is, less than 250 μs). Consequently, there will be a relatively long idle time during the congestion-detection phase, resulting in poor performance. Finally, in the Partition/Aggregate traffic pattern, each receiver is required to wait for responses from all the servers to form a meaningful result. Therefore, the overall performance is determined by the congested connection even though most of the connections do not experience any packet loss [7][8][9][10].

Transport protocols have been developed to address these problems [4][5][6][7][8][9][10], and we categorize them into three classes: ECN-based, RTT-based, and switch-based approaches. The Explicit Congestion Notification (ECN)-based approach [7][10] utilizes the ECN [13] functionality in an end-to-end manner to throttle the flows in proportion to the congestion, thus reducing queuing delay and packet drops. However, these approaches are not scalable in terms of the number of concurrent TCP connections; they can manage only a few tens of connections while the number of concurrent flows in actual data centers can exceed 40 (the maximum number used in [7]). The RTT-based approach exploits the RTT as an indication of the queuing delay, i.e., the network congestion. Unlike the Internet, the RTT in data center networks is very unreliable. The typical RTTs in data center networks are in the microsecond granularity meaning that small spikes in the RTT measurements could lead to miscalculations by the algorithm [7][8]. Moreover, it is not easy to obtain a stable average or minimum RTT, as the flow sizes for Partition/Aggregate applications are generally too small. The switch-based approach [11][12] employs functions on the switch to reduce the TCP delay, but generally suffers from deployment issues, such as high cost, long turn-around time, and backward compatibility [10].

In this paper, we propose FaST, a novel, fine-grained congestion-control algorithm whose main objective is to provide scalability to the data center network. We introduce four key contributions that address the aforementioned challenges. First, we reveal the scalability limits of the legacy approaches through analysis and observe that the congestion control should be fine-grained. FaST manipulates the segment size to render the fine-grained congestion control.

Other features of the legacy TCP such as slow start, congestion avoidance, and fast recovery are retained for backward compatibility. Second, FaST employs a novel, virtual congestion window (vwnd) to conduct the congestion-control algorithm. Third, we implement FaST on the ns-3 simulator and show that FaST outperforms the legacy algorithms. Furthermore, FaST can manage 50 concurrent flows without packet losses and up to 100 flows with at most one TCP timeout per flow. Finally, FaST is easy to deploy since it requires only a few modifications at the server-side.

The remainder of this paper is organized as follows. In Section 2, we review today's cloud data center communication patterns and previous work related to data center protocols. Then, we analyze the limitations of the previous approaches and discuss the motivation of our work in Section 3. In Section 4, we describe our congestion-control algorithm, FaST, that addresses the scalability issue. Section 5 presents the experimental evaluation of FaST and compares our results with NewReno and DCTCP. Finally, we conclude this paper in Section 6.

2. Background

In this section, we briefly review the Partition/Aggregate applications that run on cloud data center networks and some existing data center protocols for the incast congestion.

2.1. Cloud Applications and Partition/Aggregate Design Pattern

Many cloud applications including MapReduce [1], social networking [2], and recommendation systems [3] require high performance computing or large storage resources provided by multiple servers in a data center. To efficiently process the client's request and meet their demands, the applications usually follow the Partition/Aggregate design pattern shown in **Fig. 1**. In this design pattern, there are two types of servers: aggregators and workers. The workers provide computing power and data according to the user's requests. The aggregators gather the response data from the numerous workers. In other words, the request is *partitioned* and distributed to the workers and the results are *aggregated* by the aggregator(s) in each layer.

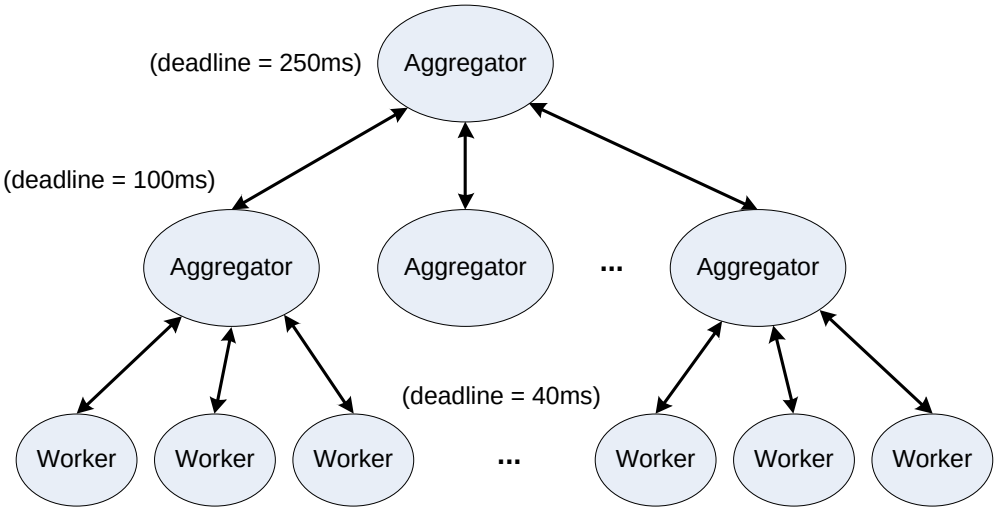

Fig. 1. Typical Partition/Aggregate design pattern in cloud applications

If these applications are extended to *online* cloud services, there are stringent delay requirements for the service to complete; there could be a Service Level Agreement (SLA) between the service providers and users. In this case, the results must be delivered within the SLA, typically 200-300 ms in a data center [7]. To meet this deadline, the workers may have deadlines of only a few tens of milliseconds as shown in **Fig. 1**. Note that if the query completion time is longer than the deadline, the results could be discarded. This not only affects the quality of service for the users but also results in a significant decrease in the operator revenue [7][11]. Therefore, it is very important to reduce the query completion time in today's data center communications.

2.2. Data Center Protocols for Incast Congestion

As described in Section 1, TCP incast congestion is one of the main causes of poor performance (for example, long query completion time). It generally causes multiple packet losses and TCP timeouts, forcing the client to be idle for RTO. To mitigate this incast congestion, the early solutions were (i) reducing client's receive socket buffer size below 64 KB, (ii) reducing the duplicate-ACK threshold, and (iii) disabling slow-start to avoid retransmission timeout [4]. However, these approaches do not fundamentally address the incast congestion. V. Vasudevan et al. [5] proposed a safe and effective fine-grained retransmission timeout value by reducing the minimum RTO from millisecond to microsecond-granularity. This approach is effective for long-term flows achieving high goodput, but it is demonstrated that retransmission timeouts less than 10 ms can cause spurious retransmission (that is, false alarms of loss detection) [12].

Even though the solutions described above improve TCP throughput, they have only focused on cluster-based storage systems as an application. By monitoring production traffic from a 6000-server, data center cluster, M. Alizadeh et al. [7] found that there are several types of applications in data centers and small query traffic generated by soft, real-time applications can experience long queuing delay because of large background flows. To maintain low buffer occupancy at the ToR switches, they proposed DCTCP that provides ECN-based congestion window control. However, DCTCP still suffered from incast congestion when the number of workers was more than 35 in their experimental environment [7]. Another congestion avoidance approach for data center networks is Incast-congestion Control for TCP (ICTCP) [8]. In [8], the authors suggested that RTT is not a good congestion indicator in high-bandwidth and low-latency networks such as data center environments. For this reason, ICTCP measures the bandwidth of the total incoming traffic and controls the receive window of each connection such that the total traffic is less than the link capacity. The incast congestion, however, can occur if the number of workers is extremely large even when the window size of each server is one. Similarly, IA-TCP [9] controls the workers' sending rate such that it does not exceed the bandwidth-delay product at the receiver side. It is more scalable in terms of the number of concurrent flows. However, it does not solve the network congestion that occurs at the Aggregation/Core level switches since it assumes that the bottleneck is only at the edge ToR switches.

While the previous studies described above are host-based approaches, switch-based solutions are also proposed. D^3 [11] performs explicit rate control in a centralized manner at the data center switches, to allocate bandwidth based on each flow's deadline and size. DeTail [12] is an in-network multipath-aware congestion-control mechanism and takes a traffic engineering-based approach to reduce the flow-completion time tail. However, these switch-based solutions require high-cost and/or customized hardware chips in the network. This is a definite hurdle for deployment.

3. Analysis and Motivation

We consider a typical Partition/Aggregate application-data center network topology where an aggregator communicates with several workers as shown in **Fig. 2**. In the Partition/Aggregate application, each worker simultaneously sends a specific size response data to the aggregator. The congestion, therefore, usually occurs at the ToR switch buffer adjacent to the Aggregator. Consequently, the DCTCP performs its congestion control at the worker side. In this section, we briefly describe the main algorithm of DCTCP and analyze the DCTCP performance in terms of scalability.

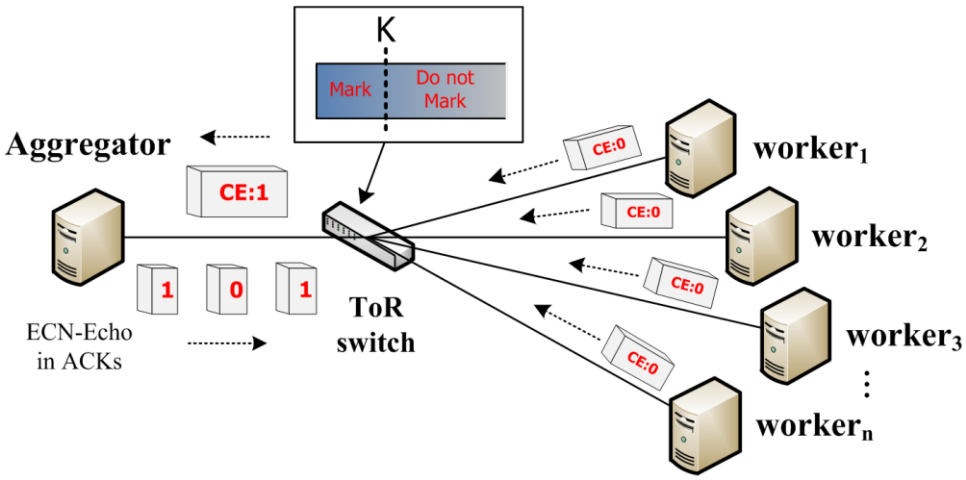

Fig. 2. Operation of DCTCP on a typical data center network topology

The key notations are summarized in Table 1.

Table 1. Key notations

Notation	Definition
α	Extent of network congestion ($0 \leq \alpha \leq 1$)
B	Switch buffer size
K	Switch buffer threshold to mark with CE bits
W^*	Window size of at which the queue size reaches K
W_{min}	Minimum window size
N^*	Number of flows of at which all the flows' window sizes reaches W_{min}
N^*_{max}	Number of flows of at which the queue size reaches B

3.1. Congestion Avoidance of DCTCP

DCTCP conducts conservative congestion control by adapting to the extent of the congestion using the ECN feedbacks. More specifically, DCTCP counts the number of ECN-marked packets and reduces the congestion window (cwnd) in proportion to the fraction of the ECN-marked packets. The legacy TCP simply halves the cwnd in response to the ECN feedback. The ECN-enabled switch marks an arriving packet with the Congestion Encountered (CE) bit if the current queue occupancy exceeds the threshold K as shown in **Fig. 2**. This

CE-bit feedback is echoed to the sender (that is, the worker) through the corresponding ACK packet. Next, for each RTT, the sender calculates the fraction of packets that were marked in the last window, F, as follows:

$$F = \frac{Number\ of\ marked\ ACKs}{Total\ number\ of\ ACKs} \tag{1}$$

The extent of the congestion, α, is obtained as follows:

$$\alpha \leftarrow (1-g)\times\alpha + g\times F \tag{2}$$

where g is a weight ($0 < g < 1$). With this information, the cwnd is updated as follows:

$$cwnd \leftarrow \max\left(cwnd\times(1-\alpha/2), 2\right) \tag{3}$$

Thus, if all the packets are marked, leading α to be close to 1 (high congestion), DCTCP will reduce the cwnd by almost half, the same as TCP. On the other hand, when α is close to 0 (low congestion), the cwnd will be only slightly reduced.

3.2. Scalability of DCTCP

We first look at the critical window size of DCTCP flows, W^*. This is the window size at which the queue size reaches K as defined in [7]. Assuming that there are N DCTCP flows whose window sizes are synchronized with the identical RTT, the critical window size is expressed as:

$$W^* = \frac{C\times RTT + K}{N\times MSS} \tag{4}$$

where C is the capacity of the bottleneck link shared by the N flows, and MSS is the Maximum Segment Size. Once the flows' window size reaches W^*, it will be reduced within a few RTTs as the switch begins to mark the data packets. This implies that W^* indicates the maximum window size before causing ECN-marking. It is eventually converged to the minimum cwnd as N increases to a large number. Now, we define N^* to be the number of flows that reduces all the flows' window sizes to the minimum cwnd (W_{min}). From (4), it follows that:

$$N^* = \frac{C\times RTT + K}{W_{min}\times MSS} \tag{5}$$

If $N \geq N^*$, the window size of each sender should be as large as W_{min}, and no window modification can be further applied. We found that N^* is not a high value in practice. For example, let us suppose that the link capacity is 1 Gbps and RTT is 100 μs. The typical value of K is 20 (packets) while W_{min} is 2 according to (3). In this case, N^* would be about 14 if the MSS is 1.5 KB.

Next, we consider a case where the total number of outstanding packets is just before the buffer overflows. It follows that:

$$\sum_{i=1}^{N} W_i \times MSS = C \times RTT + B \tag{6}$$

where W_i is the cwnd of the ith connection and B is the buffer size at the switch. We define N^*_{max} to be the maximum number of flows that satisfies (6) when $N \geq N^*$. Then, we have:

$$N^*_{max} = \frac{C \times RTT + B}{W_{min} \times MSS} \tag{7}$$

If we apply the above example to (7), the value of N^*_{max} is about 37.5 when B is 100 KB. This number corresponds directly to the result conducted in [7] that shows that the incast congestion occurs when the number of servers is larger than 35. In other words, N^*_{max} indicates the scalability of DCTCP in terms of the number of concurrent TCP flows.

Equation (7) suggests that there are options to increase N^*_{max} by controlling the parameters. Among the available parameters, however, the MSS may be the only controllable parameter since C, B, and RTT are uncontrollable; C and B depend on the switch hardware capacity and the RTT depends primarily on the network topology. Furthermore, there is not much room to control W_{min} as its typical value is either 1 or 2 in most settings. Therefore, we attempt to control the parameter MSS and we realize this using fine-grained congestion control through a *virtual congestion window*.

4. FaST: Fine-grained and Scalable TCP

In this section, we explain our proposed algorithm, called FaST, that is scalable for cloud applications that utilize a large number of workers. Our algorithm achieves scalability through fine-grained congestion control. FaST utilizes ECN and calculates α at the server side as we did in Eq. (2). This is similar to DCTCP. The main difference is that we adjust the current segment size to increase N^*_{max}. For this reason, we employ a *virtual congestion window* (*vwnd*) that can be less than W_{min} in the window update procedure (3). The *vwnd* replaces the legacy TCP *congestion window*. Whenever ECN-marked packets are observed, the *vwnd* is updated as follows:

$$vwnd \leftarrow vwnd \times (1 - \alpha / 2) \tag{8}$$

Unlike DCTCP, *vwnd* can scale below W_{min} to adjust the segment size (*SegmentSize*):

$$SegmentSize = \begin{cases} MSS, & vwnd \geq 1 \\ MSS \times vwnd, & vwnd < 1 \end{cases} \tag{9}$$

The segment size is reduced below 1 MSS if the *vwnd* is smaller than 1, otherwise it is maintained at the MSS. Note that reducing the segment size may degrade TCP performance as it increases the total number of RTTs to complete the flow, resulting in slower flow completion time. We cap the minimum segment size to 100 Kbytes because the default TCP and IP headers are 40 Kbytes in size.

The basic thought is that it is efficient to use the default MSS when N is small enough (that is, $N < N^*$), but we should reduce the segment size in proportion to the extent of the congestion

when $N \geq N^*$. Since DCTCP estimates neither N nor N^*, the *vwnd* is used instead, as it will eventually be less than W_{min} when $N \geq N^*$.

Algorithm 1. FaST sender-side algorithm.

```
 1: For the first m data packets – Probing Stage:
 2: vwnd ← 0.5
 3: SegmentSize ← Default * vwnd
 4:
 5: Initialization after m data packets:
 6: vwnd ← W_min
 7: SegmentSize ← Default MSS
 8:
 9: On observing ECN-marked ACKs:
10: update vwnd          // Equation (8)
11: update SegmentSize   // Equation (9)
12:
13: On sending data packets:
14: if vwnd < W_min then
15:    limit = max(vwnd, 1) * SegmentSize
16:    Allow to send data as much as limit (bytes) in each round
17: endif
```

Algorithm 1 presents our FaST algorithm conducted at the sender. The FaST session begins the Probing Stage that carefully starts with small segments (lines 1-3) for the first m (≤ 3) data packets because there is no history about the current network condition. For example, the current link can be saturated by a large number of flows, or a Partition/Aggregate type application may have tens or even hundreds of workers simultaneously sending packets. This probing stage can manage a large number of concurrent flows generated in a short time. Then, we initialize the *vwnd* and segment size to their default value (lines 5-7). Whenever the sender observes the ECN-marked ACKs, it updates the *vwnd* and segment size (lines 9-11) according to (8) and (9). Lines 13-17 implement our scalable flow control. If the *vwnd* is less than W_{min} (this implies $N \geq N^*$), we adjust the amount of sending data with the reduced segment size to effectively mitigate network congestion. Otherwise, our scheme generally works similar to ECN-enabled TCP, for example, DCTCP; the basic schemes such as additive increase, fast retransmit, and congestion avoidance operate similarly.

Finally, we realized that most TCP implementations could have a problem avoiding the Silly Window Syndrome (SWS) [16] with such a small data packet size. To address this situation, we must turn off the SWS function. In principle, the SWS function is used for general applications to avoid very small sending windows. However, we turn this function off only for the case when the Partition/Aggregate cloud applications are used. Turning the SWS function off can easily be achieved at the sender-side by disabling the Nagle algorithm through the TCP_NODELAY option. Therefore, other applications such as background transfers will not be affected by this action, not causing any problems in practice.

5. Evaluation

For the evaluation, we implemented the FaST algorithm in an ns-3 simulator [15]. In this section, we first describe our simulation setup and evaluate the performance in terms of the query completion time, number of packet losses, throughput, and fairness.

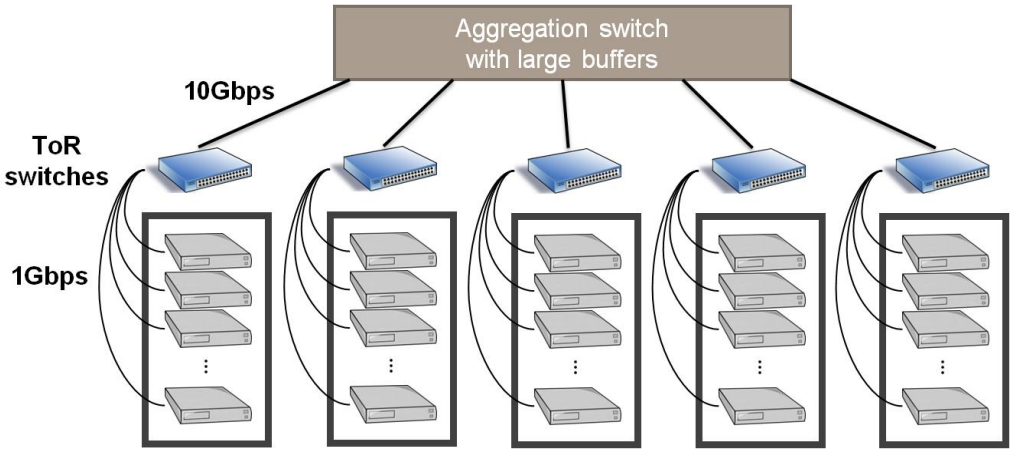

Fig. 3. Simulation network topology

5.1. Simulation Setup

Fig. 3 depicts our simulation network topology that consists of one aggregation switch, five Top-of-Rack (ToR) switches, and 20 servers per rack, for a total of 100 servers. The link rate is 1 Gbps for the ToR switches and 10 Gbps for the aggregation switch. The packet buffer size per port is set to 100 Kbytes for the ToR switches assuming that they are shallow-buffered commodity switches [7]. We deploy a large buffer for the aggregation switch. The link delay is set to 25 μs, and hence, the longest round-trip propagation delay is about 200 μs, a commonly acceptable value in today's data center networks [7].

We compare performance between FaST and two existing schemes, NewReno and DCTCP. For the key parameters of DCTCP and FaST, we set g, the weighted averaging factor, to 1/16. We set K, the buffer occupancy threshold for marking CE bits, to 20 packets for the 1 Gbps links and 65 packets for the 10 Gbps according to [7]. The RTO_{min} for all the TCP senders is set to 10 ms.

To emulate typical cloud applications, we developed a Partition/Aggregate application that consists of one root and n workers. The root sends a query to its workers and each worker responds with the requested amount of data. After all the response data are received from the workers, the query completion time is measured. The measurements are repeated 100 times in all simulations.

5.2. Performance with no Background Traffic

In this subsection, we increase the number of workers, n, from 10 to 100. We set the response data size of each worker to 10 KB as the size of the Partition/Aggregate flows is only a few KBs in general cloud data center networks [7][17].

We first measure the average query completion time as shown in **Fig. 4**. When the number of workers is small (that is, 10 to 30), the response data is transmitted to the root quickly

without any packet losses resulting in low query completion times for all the protocols. However, as the number of workers increases, NewReno and DCTCP experience multiple packet losses and their query completion time increases to approximately 72 ms and 43 ms, respectively. On the other hand, FaST shows a very low completion time until the number of workers reaches 50 and takes only 16 ms even with 100 workers.

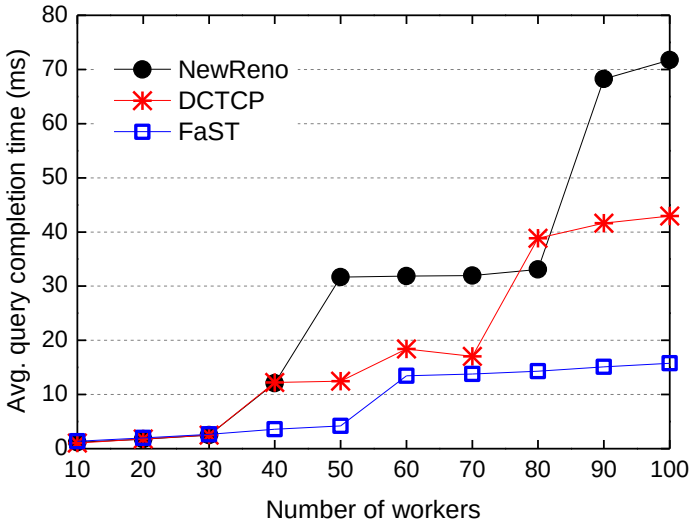

Fig. 4. Average query completion time

Fig. 5 shows the average number of total TCP timeouts. As observed in **Fig. 4**, there are no timeouts when the number of workers is 10 to 30. Overall, the average number of total TCP timeouts linearly increases in proportion to the number of workers, but FaST performs much better than NewReno and DCTCP, causing a smaller number of TCP timeouts. Therefore, we confirm that FaST effectively mitigates the network congestion at the bottleneck port by adjusting the segment size.

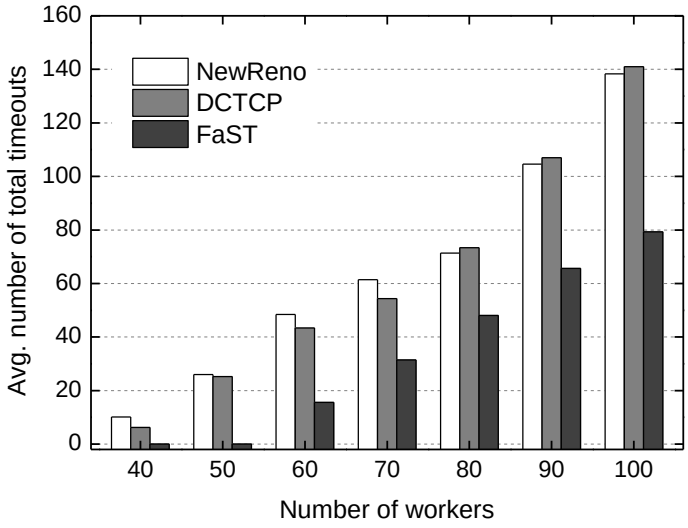

Fig. 5. Average number of total TCP timeouts

To investigate the effect of multiple TCP timeouts on the query completion time, we depict the cumulative distribution function CDF of the number of TCP timeouts of each worker when the number of workers is 100 as shown in **Fig. 6**. In this figure, it is observed that more than 50% of NewReno and DCTCP flows experience multiple TCP timeouts, and in the case of NewReno, a few workers suffer three consecutive timeouts, resulting in the long query completion time shown in **Fig. 4**. Finally, we observe that most FaST flows experience at most one TCP timeout.

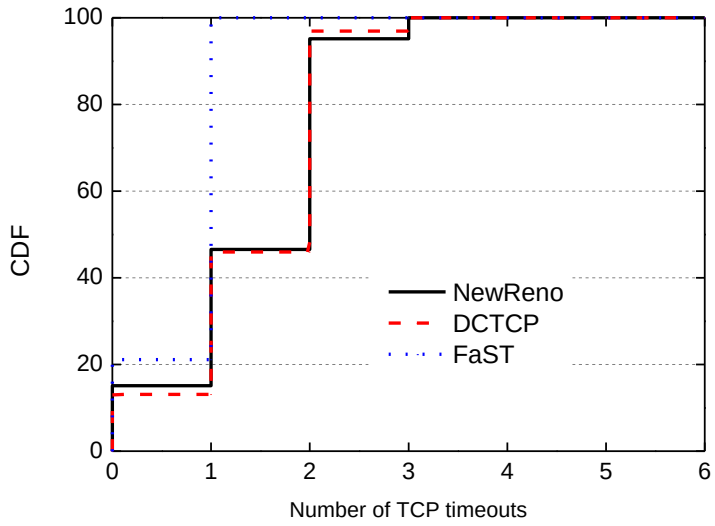

Fig. 6. CDF of the number of TCP timeouts in each worker (n = 100)

5.3. Performance with Background Traffic

Next, we add one background flow to the previous scenario. It is reported that the median number of large flows in data center networks [7] is one. The size of the background flow is 10 MB and it is directed to the same receiver (that is, the root node). This flow fully utilizes the bottleneck link before the Partition/Aggregate application begins.

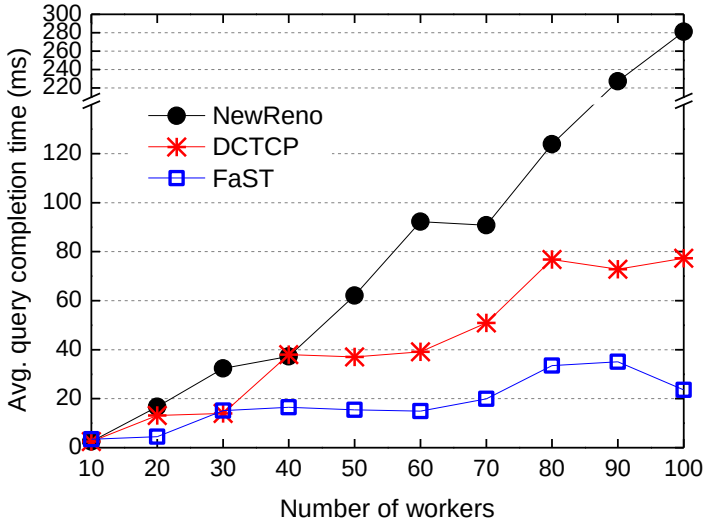

Fig. 7. Average query completion time with background traffic (10 MB)

Fig. 7 shows the average query completion time for the Partition/Aggregate application. The overall query completion times of all the protocols increase because of the effect of the background flow. The completion time of NewReno, in particular, increases up to 281 ms when the number of workers is 100. We observe that a number of NewReno flows perform several exponential backoffs as their retransmitted packets are continuously lost. This is mainly because the background flow fully utilizes the bottleneck port. On the other hand, the background flows that employ DCTCP or FaST try to keep the buffer occupancy low to provide room for short query flows. We also see that FaST achieves relatively lower completion times than DCTCP (under 35 ms).

Fig. 8 shows the average number of total TCP timeouts. As expected from **Fig. 7**, the number of total TCP timeouts increases slightly for NewReno and DCTCP compared to **Fig. 5**. FaST begins to experience TCP timeouts when the number of workers is 30, but the overall number of TCP timeouts is almost the same as **Fig. 5**.

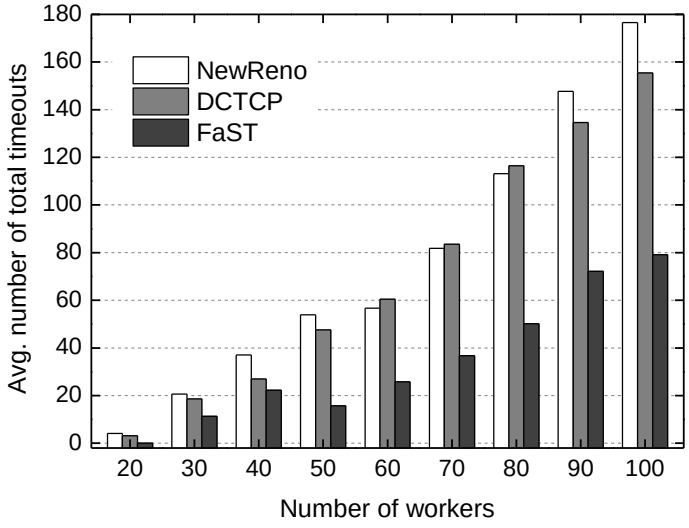

Fig. 8. Average number of total TCP timeouts with background traffic

Fig. 9 shows the average throughput of the background flow. When the number of workers is below 50, NewReno shows a high average throughput while the results of DCTCP and FaST are similar and less than that of NewReno. This occurs because the background (long-term, large) flow with NewReno easily overrides the small flows for cloud applications as explained above; hence, it can fully utilize the bottleneck link and achieve high throughput. FaST shows lower throughput than NewReno when the number of workers is less than 50, but it is still comparable to DCTCP. As the network congestion becomes severe (with more than 50 workers), we observe that the throughput of the background flow for FaST is comparable with that of NewReno.

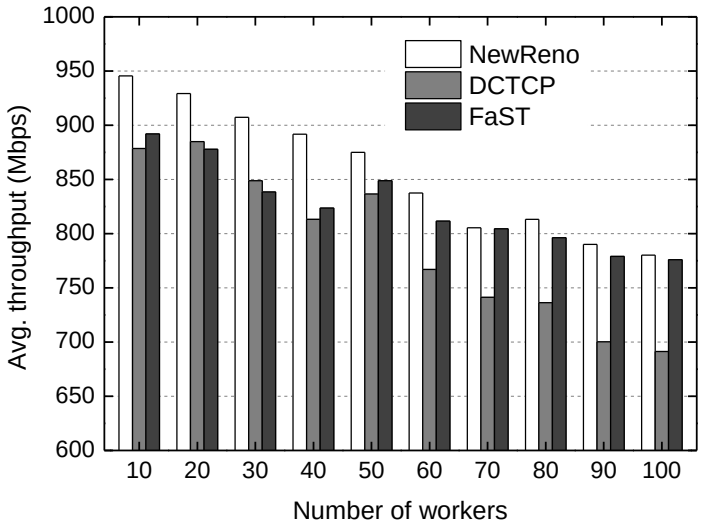

Fig. 9. Average throughput (Mbps) of the background flow

5.4. Convergence and Fairness

To confirm that FaST flows quickly converge to their fair-share, similar to DCTCP, we set up five workers under the same rack. Each worker transmits a large amount of data (1 GB) to the same receiver and starts sequentially with a 3-second interval. **Fig. 10** shows the throughput variation for the five FaST flows, and we confirm that their throughputs are quickly converged to the fair-share with this graph.

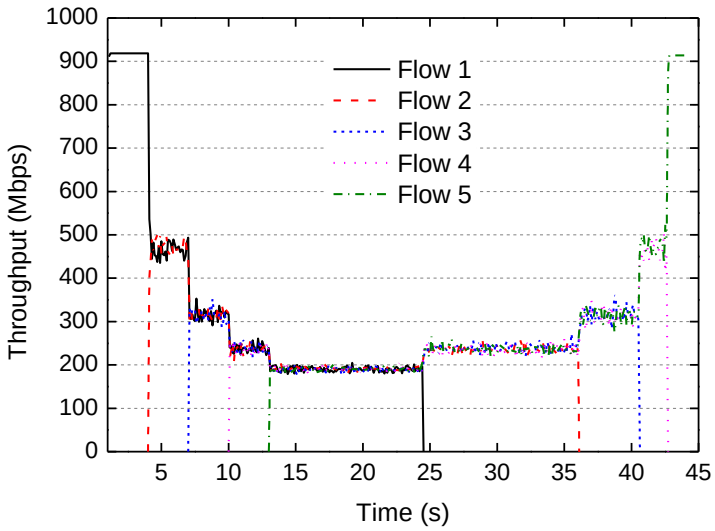

Fig. 10. Convergence test with five FaST flows

We also measure long-term throughputs for 45 workers under the same rack as shown in **Fig. 11** to observe the fairness among FaST flows.

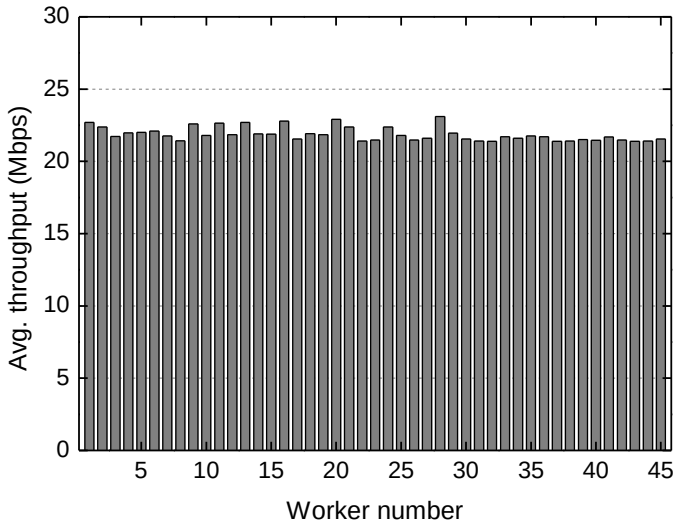

Fig. 11. Throughput fairness among 45 FaST flows

Next, we calculate Jain's fairness index [18] as follows:

$$Fairness\ index = \frac{\left(\sum_{i=1}^{n} T_i\right)^2}{n\sum_{i=1}^{n} T_i^2} \tag{10}$$

where T_i is throughput of the ith worker and n is the total number of workers. The fairness index of **Fig. 11** is 0.999, indicating that FaST flows achieve good fairness. We note that TCP friendliness (that is, inter-fairness to NewReno) is not considered in this paper because in most data centers, backward compatibility and fairness to legacy protocols are not major concerns as they are under a single administrative control [7].

5.5. Query Completion Time vs. Segment Size

The proposed algorithm dynamically adjusts the current segment size that might be smaller than MSS. However, the smaller segment size generally results in low transmission performance of TCP. Therefore, it is important to show how TCP performance is affected by different segment sizes under the Partition/Aggregate traffic pattern.

Fig. 12 shows the average query completion time of DCTCP, by increasing the segment size from 100 to 1500 bytes. It is observed that the higher segment size results in better performance (i.e., low query completion time) when the number of concurrent flows (N) is 25. However, as N increases to more than 40, DCTCP suffers from network congestion, which results in higher query completion time with larger segment sizes. In this case, the smaller segment size shows lower query completion times by avoiding congestion. For example, when N = 40, the query completion time is about 12.5 ms with 1500-byte segments. However, reducing the segment size to 1000 bytes could avoid network congestion, showing the lowest completion time. Therefore, the proper segment size should be adjusted according to the extent of the network congestion.

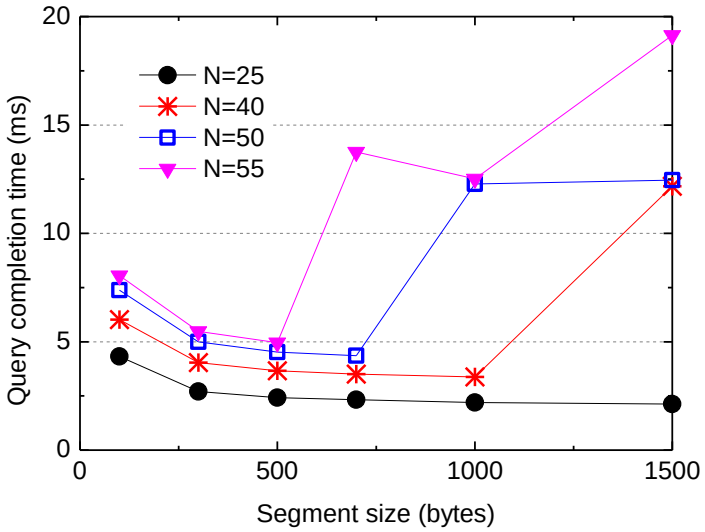

Fig. 12. Query completion time vs. Segment size

6. Conclusion

In this paper, we propose a scalable congestion-control scheme for cloud data center applications. Our analysis indicates that the packet buffer at the ToR switches can be full even with a few tens of concurrent flows. To mitigate this scalability problem, we employ a fine-grained congestion control, called FaST, using a virtual congestion window. By doing this, we achieve low query completion times for the short flows generated by cloud applications, while still showing comparable average throughput for background traffic. In addition, this approach is simple to implement and the actual deployment is easy, as it requires only a small modification at the server-side. Through ns-3 simulations, we confirm that the proposed scheme manages well with 50 concurrent flows without packet losses and 100 flows with at most one TCP timeout per flow. It achieves these results while outperforming other legacy data center congestion-control protocols.

As future work, we plan to construct a data center testbed with ECN-supported switches and perform real experiments to verify the performance of the proposed algorithm.

References

[1] J. Dean and S. Ghemawat, “MapReduce: Simplified Data Processing on Large Clusters,” in *Proc. of USENIX OSDI*, 2004. Article (CrossRef Link).

[2] D. Beaver, S. Kumar, H. C. Li, J. Sobel, and P. Vajgel, “Finding a Needle in Haystack: Facebook’s Photo Storage,” in *Proc. of USENIX OSDI*, 2010.

[3] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, “Dynamo: Amazon’s Highly Available Key-value Store,” in *Proc. of ACM SOSP*, 2007. Article (CrossRef Link).

[4] A. Phanishayee, E. Krevat, V. Vasudevan, D. G. Anderson, G. R. Ganger, G. A. Gibson, and S. Seshan, “Measurement and Analysis of TCP Throughput Collapse in Cluster-based Storage Systems,” in *Proc. of USENIX FAST*, 2008.

[5] Y. Chen, R. Griffith, J. Liu, R. H. Katz, and A. D. Joseph, “Understanding TCP Incast Throughput Collapse in Datacenter Networks,” in *Proc. of ACM WREN*, 2009. Article (CrossRef Link).

[6] V. Vasudevan, A. Phanishayee, H. Shah, E. Krevat, D. G. Anderson, G. R. Ganger, G. A. Gibson, and B. Mueller, "Safe and Effective Finegrained TCP Retransmissions for Datacenter Communication," in *Proc. of ACM SIGCOMM*, 2009. Article (CrossRef Link).
[7] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data Center TCP (DCTCP)," in *Proc. of ACM SIGCOMM*, 2010. Article (CrossRef Link).
[8] H. Wu, Z. Feng, C. Guo, and Y. Zhang, "ICTCP: Incast Congestion Control for TCP in Data Center Networks," in *Proc. of ACM CoNEXT*, 2010. Article (CrossRef Link).
[9] J. Hwang, J. Yoo, and N. Choi, "IA-TCP: A Rate Based Incast-Avoidance Algorithm for TCP in Data Center Networks," in *Proc. of IEEE ICC*, 2012. Article (CrossRef Link).
[10] B. Vamanan, J. Hasan, and T. N. Vijaykumar, "Deadline-Aware Datacenter TCP (D^2TCP)," in *Proc. of ACM SIGCOMM*, 2012. Article (CrossRef Link).
[11] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowstron, "Better Never than Late: Meeting Deadlines in Datacenter Networks," in *Proc. of ACM SIGCOMM*, 2011. Article (CrossRef Link).
[12] D. Zats, T. Das, P. Mohan, and R. Katz, "DeTail: Reducing the Flow Completion Time Tail in Datacenter Networks," in *Proc. of ACM SIGCOMM*, 2012. Article (CrossRef Link).
[13] K. Ramakrishnan, S. Floyd, and D. Black, "The addition of explicit congestion notification (ECN) to IP," *RFC 3168, IETF*, 2001.
[14] D. Nagle, D. Serenyi, and A. Matthews, "The Panasas ActiveScale Storage Cluster – Delivering Scalable High Bandwidth Storage," in *Proc. of ACM/IEEE SC2004 Conference*, 2004. Article (CrossRef Link).
[15] The ns-3 discrete-event network simulator. [Online]. Available: http://www.nsnam.org/.
[16] D. D. Clark, "Window and Acknowledgement Strategy in TCP," *RFC 813, IETF*, Jul. 1982.
[17] T. Benson, A. Akella, and D. A. Maltz, "Network Traffic Characteristics of Data Centers in the Wild," in *Proc. of* ACM IMC, 2010. Article (CrossRef Link).
[18] R. Jain, "The Art of Computer Systems Performance Analysis," *John Wiley & Sons*, 1991.

Jaehyun Hwang received the B.S. degree in computer science from Catholic University of Korea, Seoul, Korea in 2003, and the M.S. and Ph.D. in computer science from Korea University, Seoul, Korea in 2005 and 2010, respectively. His research backgrounds are mainly in TCP, focusing on a flexible TCP structure, advanced TCP flavors and their performance. Since September 2010, he has been with the networking research domain at Bell Labs, Alcatel-Lucent, Seoul, Korea as a Member of Technical Staff. His current research interests include data center networks, software-defined networking, multipath TCP, and HTTP adaptive streaming.

Joon Yoo received his B.S. in Mechanical Engineering from KAIST, and Ph.D in Computer Science and Engineering from Seoul National University in 1997 and 2009, respectively. He worked as a postdoctoral researcher at the University of California, Los Angeles from 2009 to 2010, and then he worked at Bell Labs, Alcatel-Lucent, Seoul, Korea as a Member of Technical Staff from 2010 to 2012. He has been with the Department of Software Design & Management, Gachon University, Korea, as an assistant professor since 2012. His research interests include vehicular networks, data center networks, and IEEE 802.11 WLAN.