SDN-based TCP Congestion Control in Data Center Networks

Yifei Lu, Shuhong Zhu
School of Computer Science and Engineering
Nanjing University of Science and Technology
Nanjing, China
luyifei@njust.edu.cn

***Abstract*—TCP incast usually happens when a receiver requests the data from multiple senders simultaneously. This many-to-one communication pattern constantly appears in the data center networks due to the data are stored at multiple servers. With Software Defined Networks (SDN), the centralized control methods and the global view of the network can be an effective way to handle this problem. In this paper, we propose a SDN-based TCP (SDTCP) congestion control mechanism at network side. Our approach enables controller to select a long-lived flow to reduce sending rate by adjusting the TCP receive window of ACK packet after OpenFlow-switch triggered a congestion message to controller. The key benefit of SDTCP is that, with global perspective, we can accurately decelerate the rate of long-lived flow to ensure the performance other flows. The experiments indicate that we can achieve almost zero packet loss for TCP incast and guarantee goodput for the high propriety flows.**

Keywords—Data center networks; TCP; Incast; Congestion control; SDN

I. INTRODUCTION

Data centers are becoming one of hottest topics in both research communities and IT industry. In today's data center networks, TCP has been used as the de facto transport layer protocol to ensure reliable data delivery. However, the unique workloads scale and environments of the data center work violate the WAN assumptions on which TCP was originally designed. A reported open problem is TCP incast [1, 2] .

TCP incast problem was initially identified in distributed storage cluster [1] and has nowadays become a practical issue in data center networks. TCP incast, which results in gross under-utilization of link capacity, occurs in synchronized many-to-one communication patterns. Fig. 1 shows a typical TCP incast scenario used by many literatures. In such a communication scenario, a receiver issues data requests to multiple senders. The senders respond to the request and return an amount of data to the receiver. The data from all senders pass through a bottleneck link in a many-to-one pattern to the receiver. When the number of synchronized senders increases, throughput observed at the receiver drops to one or two orders of magnitude below the link capacity.

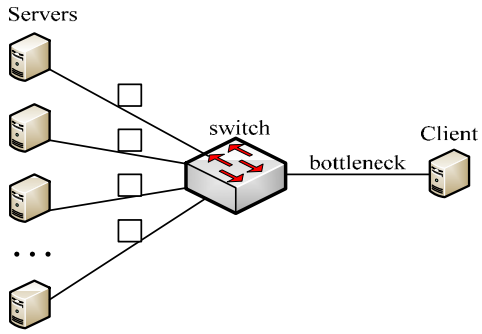


Fig. 1. TCP incast scenario

Data centers provide resources for a broad range of services, such as web search, email, web sites, etc., each with different delay requirements. For example, web search should cater to users' requests quickly, while data backup has no special requirement on completion time. In order to design efficient transmission mechanisms for data centers, the analysis of data center traffic characteristics is important [3, 4]. The flows in a data center networks tend to be bursty and they are either large data flows that require high throughput and delay-insensitive (background flows) or short control or web search flows that require low delay (bursty flows) [5]. TCP incast problem would severely degrade the application performance especially those bursty and delay-sensitive communications such as MapReduce [6], Dryad [7], and large-scale partition/aggregate web applications [2, 8]. In this paper, we presume background and bursty flows coexist in data center networks.

Previous solutions, focused on either reducing the waiting time for packet loss recovery by faster retransmissions [9], or controlling switch buffer occupation to avoid overflow by using ECN and modified TCP at both sender and receiver sides [2]. These solutions need to modify the TCP at end-system and does not consider the characteristics of flow.

Software Defined Networking (SDN) [10] is a revolutionary network architecture that separates out network control functions from the underlying equipment and deploys them centrally on the controller, where OpenFlow is the standard interface. This technique enables the network

978-1-4673-8590-9/15/$31.00 ©2015 IEEE

administrators and the application developers to dynamically manage and alter network parameters in runtime and according to the current demand [11]. Unique characteristics of SDN have made it an appropriate choice for data center network and in particular suited for cloud networking [12].

An important capability of SDN is the fact that it enables applications to be aware of the network topology and congestion. In this paper, we use this feature of SDN to design and implement an innovative SDN-based TCP congestion control mechanism (SDTCP). The key design principle behind SDTCP is that we avoid network congestion by adjusting the TCP receive window of ACK packet at controller to reduce sender's transmission rate. We employ a very simple queue management scheme in OpenFlow-switch (OF-switch) that will trigger a congestion notification to controller when queue occupancy is greater than threshold K. When controller receives this signal, it will select a long-lived flow and push a modify command to modify receive window of ACK packet at OF-switch automatically.

The rest of the paper is organized as follows. Section II discusses related works. Section III describes the design rationale of SDTCP. Section IV shows the implementation of SDTCP. Section V presents experimental results. Finally, Section VI concludes the paper.

II. RELATED WORKS

Since the TCP incast problem was proposed by Nagle et al. in [1], some work has been done to address it. It is well known that plenty of existing work with respect to TCP congestion control has been proposed so far. In this section, we just summarize the most relevant works. We discuss two kinds of existing solutions to TCP incast problem: window-based solutions and recovery-based solutions respectively.

Window-based Solutions

The window-based solutions such as DCTCP [2] and ICTCP [13] have been proposed to mitigate TCP incast congestion. The key idea of those window-based solutions is to adjust the congestion or receive window to control inflight traffic, so that it will not overfill the switch buffer.

DCTCP aims to ensure low latency for short flows and good utilization for long flows by reducing switch buffer occupation meanwhile minimizing buffer oscillation. In DCTCP, ECN with thresholds modified is used for congestion notification, while both TCP sender and receiver are slightly modified for a novel fine grained congestion window adjustment. Reduced switch buffer occupation can effectively mitigate potential overflow caused by incast. D^3 [8] uses explicit rate control to apportion bandwidth according to flow deadlines. Given a flow's size and deadline, source end hosts request desired rates to switches. The switches assign and reserve allocated rates for the flows. ICTCP [13], on the other hand, adaptively adjusts the receive window on the receiver side to throttle aggregate throughput. ICTCP measures available bandwidth and per-flow throughput in each control interval. It only increases the receive window when there is enough available bandwidth and the difference of measured throughput and expected throughput is small. Foremost, ICTCP fails to work well if the bottleneck is not the link that connects to the receiver.

Recovery-based Solutions

Unlike window-based solutions, recovery-based solutions address incast congestion via reducing the impact of timeouts.

In [14], several trials have been made to avoid TOs, such as reducing the duplicate ACK threshold of entering Fast Retransmission (FR) from 3 to 1, disabling slow start phase, and trying different TCP versions. V. Vasudevan et al [9] suggested reducing RTOmin to alleviate the goodput decline and uses high-resolution timers to enable microsecond-granularity TCP timeouts. In this way, TCP can retransmit lost packets quickly without leaving the link idle for a long time. CP proposed in [15] simply drops a packet's payload at an overloaded switch and uses a SACK-like ACK mechanism to achieve rapid and precise notification of lost packets.

OpenTCP [16] is presented as a system for dynamic adaptation of TCP based on network and traffic conditions in Software-Defined Networks (SDNs). OpenTCP is not a new variation of TCP. Instead, it complements previous efforts by making it easy to switch between different TCP variants automatically (or in a semi-supervised manner), or to tune TCP parameters based on network conditions. For instance, one can use OpenTCP to either utilize DCTCP or CUBIC in a data center environment. The decision on which variant to use is made in advance through the congestion control policies defined by the network operator.

These protocols discussed above have suffered from limited deployability, as most of them require custom modifications to switches, end-hosts, or even both. Consequently, these protocols have seen little practical use, arguably due to their difficulty of deployment.

The major difference of our work with theirs is that our target is to avoid packet loss, while they focus on how to mitigate the impact of packet loss, either less frequent timeouts or faster retransmission on timeouts. This makes our work complementary to previous work. While our focus is congestion avoidance in software defined networks to prevent packet loss in TCP incast, which is not considered in previous work.

III. DESIGN OF SDTCP ALGORITHM

A. Basic Idea

The goal of SDTCP is to achieve high burst tolerance, low latency, and high throughput, in software defined networks. To this end, SDTCP is designed to reduce throughput of background flows to guarantee bursty flows which are usually more important.

SDTCP uses a simple queue management scheme at OF-switches that trigger a congestion notification, referred to as OFPT_BUF_ALARM message in extended OpenFlow protocol, to controller as soon as the buffer occupancy exceeds a fixed threshold. This notification message which contains

queue length and OF-switch identify and port information is forwarded to controller via OpenFlow channel.

When receiving the notification message, the controller selects a long-lived flow and pushes a modify command to OF-switch, which will react by reducing the receive window of ACK packets in response to the long-lived flow. In this procedure, the controller is responsible for calculating a specific value for receive window of ACK packet.

Subsequently, sender will decrease the send window which will result in reducing transmission rate after gaining the receive window of ACK. After that, if buffer occupancy is smaller than the threshold, a recover notification message is forwarded to controller, which reacts by pushing a recover command to OF-switch. With this new command, ACK packet will not be modified and transmission rate of long-lived flow will recover.

The process of SDTCP algorithm is shown in Fig. 2.

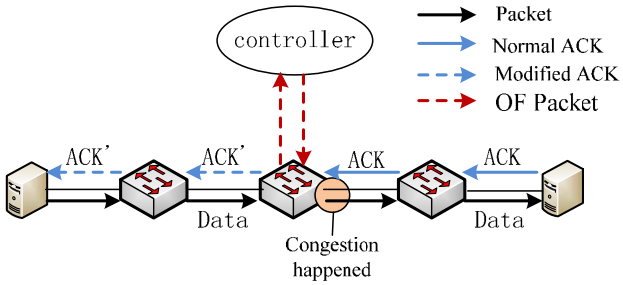


Fig. 2. The process of SDTCP

In what follows, we will mainly elaborate SDTCP mechanism we proposed by answering the following three questions.

(1) How to trigger network congestion?

(2) How to choose a suitable flow when congestion happening?

(3) How to adjust the size of receive window?

The first question is dealt with by the OF-switch and the other two are handled by controller.

B. Network Congestion Trigger

SDTCP uses tail-drop queue management which is already available in current commodity switches and Open vSwitch (OVS) [17] in our experiment. This queue management monitors the current queue size and triggers a congestion notification message when queue size is higher than a threshold value (K). This notification message, referred to as OFPT_BUF_ALARM in extended OpenFlow, contains current queue size and port rate. In the meantime, OF-switch turns into congestion state by setting variable trigger = 1 and monitors the queue length periodically with a timer. If queue size is still greater than a threshold when timer expires, congestion notification message is triggered to controller. If queue size is smaller than a threshold last for three cycles, OF-switch sends a recover notification message, named by OFPT_BUF_NORMAL, to controller. At same time, OF-switch turns off timer and sets trigger = 0 to exit the congestion state. In our paper, the value of timer is $\frac{1}{2}RTT$ so that it need at least $\frac{3}{2}RTT$ time to recover transmission rate of long-lived flow.

C. Flow Selection

When the controller receives congestion notification message from OF-switch, it will first parse the message and get useful information including OF-switch (S_i) and port (P_i) where congestion has happened. After that, controller finds flows by a rule that flows must pass through the OF-switch Si with the output port Pi. There may be several flows matching this rule and we choose a longest-lived data flow f_i . After a $\frac{3}{2}RTT$ cycle, if receiving congestion notification message again, controller will choose another longest-lived flow $f_j, j \neq i$.

D. Receive Window Adjusting

After choosing a long-live flow f_c , controller pushes a flow table matching the flow f_c with the actions for OF-switch to adjust receive window of ACK packet.

Suppose that there are *N* incast flows in the OF-switch with the same roundtrip times RTT, then we have

$$\sum_{i \in N} W_i(t) = C * RTT + Q(t) \quad (1)$$

where $W_i(t)$ is the window size of flow *i*. C is the capacity of a bottleneck link. The queue size in switch at time *t* is given by $Q(t)$.

According to equation (1), the size of current send window W_c of flow f_c, also denoted as the size of average send window, is given by:

$$W_c = \overline{W_i} = \frac{C*RTT+Q(t)}{N} \quad (2)$$

We can gain W_r, the size of receive window, and queue length $Q(t)$ when parsing the ACK packet received by controller from OF-switch. Then, it's easy to get W_c according to equation (2). Finally, we can obtain W_r', the size of receive window we need modify, as follows.

$$W_r' = max(min\left(\frac{W_c}{2}, W_r\right), MSS) \quad (3)$$

Subsequently, W_r' is updated once for OF_switch receives a new ACK as follows:

$$W_r' = max(min(W_{r\_old}' - \alpha(MSS), W_r), MSS) \quad (4)$$

where W'_{r_old} is the size of receive window of new ACK and α is a integer and $\alpha \geq 1$. We set $\alpha = 5$ in our implementation.

It is possible that all the flows are greedy flow. Hence, when the entire flows are punished by the controller, W_r' of each flow is altered as equation (5) suggested.

$$W_r' = max(min(W_c, W_r), MSS) \quad (5)$$

E. Receive Window Renewing

When receives the OFPT_BUF_NORMAL message from OF-switch, the controller extracts the information from the message and examines the database (we will discuss in Ⅳ.A) for the logs of the burst flows that using this port. Only if the controller acquires the OFPT_BUF_NORMAL message and all the FIN messages of the burst flows, the controller will renew the receiver window of the long-lived flows.

IV. IMPLEMENTATION

In this section, we discuss implementation details of SDTCP. We implement the SDTCP mechanism in Open vSwitch and Floodlight [18] with OpenFlow 1.3. We use Mininet [19] to experiment the SDN-based data center networks.

A. Flow Table Generation

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

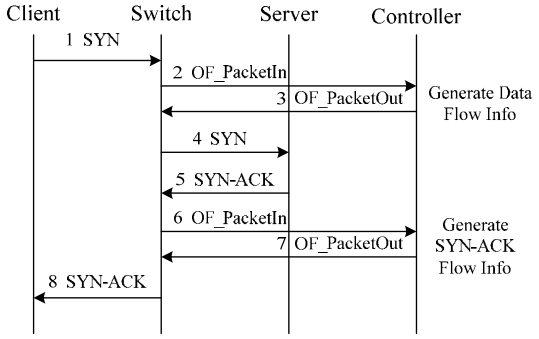


Fig. 3. TCP connection

As shown in Fig. 3, OF-switch send SYN packet to controller via Packet_in message when finding no matching entry in flow table. When receiving this Packet_In message, the controller generates routing table and pushes it to OF-switch. At same time, the controller records the information of the flow to form data flow table including source IP address, source port, destination IP address, destination port and time. Fig. 4 shows the detail of global information database.

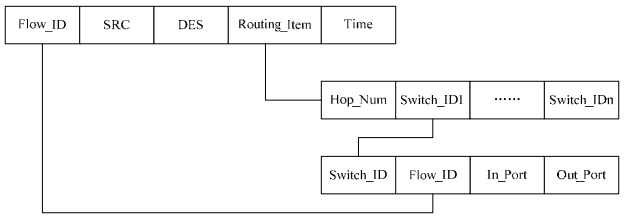


Fig. 4. Global information database

In the same way, receiver will reply a SYN-ACK packet when receiving SYN packet. This SYN-ACK packet will have the same procedure as we discussed above.

The process of connection termination is shown in Fig. 5.

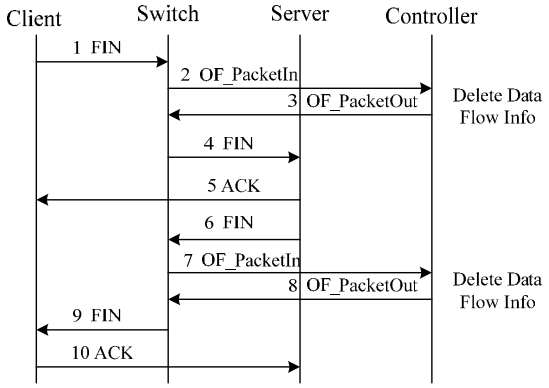


Fig. 5. TCP termination

B. OFPT_BUF_ALARM and NORMAL Packet Type

We extend the standard OpenFlow 1.3 to create the new OFPT_BUF_ALARM and OFPT_BUF_NORMAL packet, which are used to trigger the control action on the controller. The packet's fields include the standard OpenFlow header ofp_header, the ofp_port object specifying the detail of the port which the congestion happened, port_buf field indicating the buffer size of the port, the cookie field and priority field, as shown in Fig. 6. The ofp_header is described in the OpenFlow Specification and the type field in our new message is OFPT_BUF_ALARM and OFPT_BUF_NORMAL.

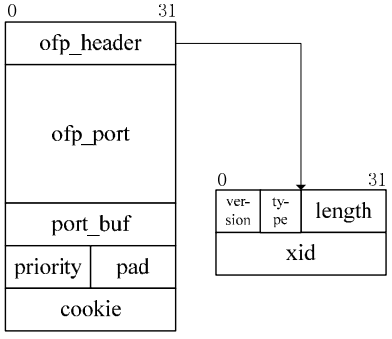


Fig. 6. OFPT_BUF_ALARM format

C. TCP_FLAGS Match Field

In the version 1.3, the TCP_FLAGS is not added into the standard Openflow protocol, but the Openvswitch 2.3.0 has already implemented the TCP_FLAGS as NXM (Nicira extensible match) for it. Therefore, we extend the protocol on the controller to create the OFPT_FLOW_MOD message with specific OXM TLV header to generate the flow entries with TCP_FLAGS match field. The OXM_OF_TCP_FLAGS is defined as NXM_1, oxm_filed:34. Hence, the 32-bit value of OXM TLV header is 0x00014402 (unmasked) or 0x00014504 (masked).

This match field can bitwise match on the TCP flags and contains two values: flags and mask. They are 16-bit numbers written in decimal or in hexadecimal prefixed by 0x. Each 1-bit in mask requires that the corresponding bit in flags must match and 0-bit means to be ignored. For example, 0x0002/0x0012 means the flow entry will match the packet with TCP SYNs which are not ACKs.

The FIN messages are handled to the controller to update the globe information database via Packet_In messages. Thus, all the switches contain a high priority flow entry to match the packets with FIN message and output it to the controller.

D. MOD_WINDOW Action

The MOD_WINDOW action is an extended subset action of the OFPAT_SET_FIELD in the Openflow 1.3 and it requires one parameter. When a flow is matched and the parameter is smaller than the original TCP window, this action will modify window size to a certain value as the parameter suggested. The parameter of the action will self-decrease by 5 MSS every time a packet matches this flow. Before pushes the flow to the switch, the controller checks the suggested TCP window to ensure it is greater than the size of 1 MSS.

Considering the window scaling mechanism, the controller will record the window scale value when the TCP connection is establishing. In the following example and other experiments, we choose 9 (real window size is the TCP window field multiply by 2^9) as the default TCP window scale value. Here is an example of using the extensional MOD_WINDOW action:

idle_timeout=10,hard_timeout=0,priority=65535,tcp,in_port=3,dl_src=3a:07:ce:24:87:3c,dl_dst=ea:0d:62:a4:e3:c3,nw_src=10.0.0.2,nw_dst=10.0.0.1,tp_src=56467,tp_dst=7000,actions=mod_window:2000,output:1

V. EXPERIMENTAL RESULTS

In this section we examine the performance of SDTCP dealing with TCP incast problem. We measure it in three experiments. First, with background flow, we examine the properties of SDTCP, such as the buff size, the single flow goodput and total goodput while comparing it with the original TCP protocol and the DCTCP. Second, with no background flow, we evaluate the SDTCP's with same properties. Finally, we design a pressure test for the controller to examine the performance of handling the large number of synchronous Packet_In packets. For DCTCP implementation, we use public code from [20] and add ECN capability to SYN packets [21].

To these ends, we conducted series experiments in the Mininet v2.2.1, using the Floodlight as the controller and the Openvswitch v2.3.0 as the OpenFlow switch. The experiments were performed on an 8 core, 2.4 GHz machine with 16 GB of RAM, and the operating system is Ubuntu 14.04.2 (kernel 3.16.0-30-generic).

Mininet is chosen for two reasons. First, Mininet serves as a real-time emulator for rapid prototyping of OF-switches. Second, Mininet is easily integrates with the OpenFlow 1.3 software switch. Our SDTCP controller is implemented on top of the floodlight platform which is an open source controller written in Java. It provides a modular programming environment so that we can easily add new modules on top of it and decide which existing modules to be run.

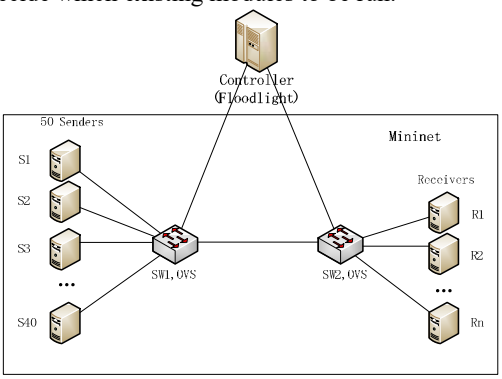
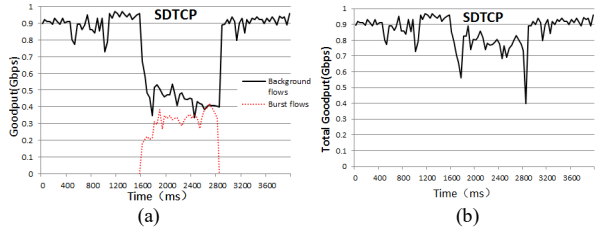


Fig. 7. Experiment topology

We deployed a testbed with 40 senders and two software switch (OVS). The topology of our testbed is shown in Fig. 7, where 40 senders connect to the SW1; multiply receivers connect to SW2 and a link between SW1 and SW2. The links have a 1Gbps throughput and a latency of 5ms each to create 30ms fixed RTT. We can know that incast congestion happens in the intermediate link. We allocate a static buffer size of 100 packets (100 × 1.5KB =150KB) to the port with congestion. When a background flow exists, it indicates that other receivers have built TCP connections with the senders before the TCP incast happened. We use TCP New Reno [22] as our congestion control algorithm and disable the delayed ACK. TCP New Reno has been used with a minRTO of 200ms, an initRTO of 3s and a Maximum Segment Size (MSS) of 1460B.

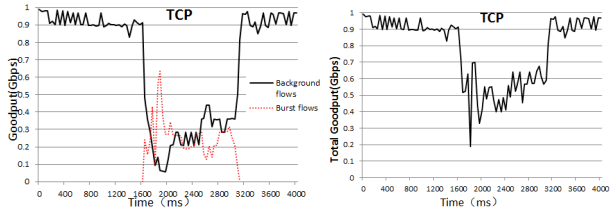
A. SDTCP performance with background flow

We establish the connections between the R2 with S11-20 as the background flows and R1with S1-S10 as burst flows. In order to trigger the SDTCP, the background flows were set to achieve almost 1Gbps of throughput. The total traffic volume of the burst flows is fixed with 50MB and both of the SDTCP and TCP are using the tail-drop queue management.



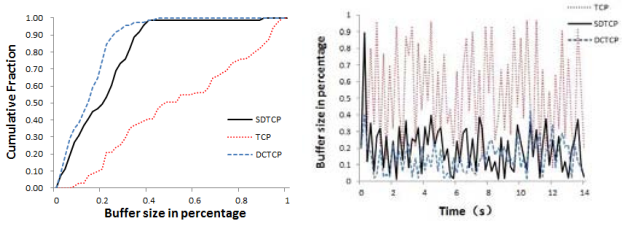
(a) (b)

Fig. 8. SDTCP flow and total goodput with background flow



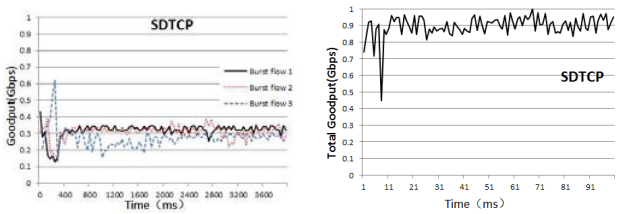
(a) (b)

Fig. 9. TCP flow and total goodput with background flow



(a) (b)

Fig. 10. Time series of buffer size and buffer size CDF



(a) (b)

Fig. 11. SDTCP flow and total goodput with no background flow

Note that both TCP and SDTCP achieve the maximum throughput of 0.97Gbps as shown in the Fig.8, 9, but the key differences are the transmission time of burst flows and the total traffic volume. Firstly, TCP consumes 1480ms to finish the transmission while SDTCP consumes 1280ms. Secondly, the TCP achieves 404MB of the total traffic volume as the SDTCP achieves 453MB.

When the burst flows arrive at a switch, the switch's buffer exceeds the threshold immediately due to the background flows. Then, the controller pushes the MOD_WINDOW flow entries to the switch to reduce the background flows' throughput as shown in the Fig.8. When the burst flows exist, the goodput of the background flow decreases constantly as we as mentioned in Subsection III.D.

We observe that TCP quickly suffer for several packet losses when the incast congestion happens. The Fig.9 (b) shows that the total goodput of the TCP is fluctuate from 0.2Gbps to 0.7Gbps while the SDTCP still maintains around 0.8Gbps as shown in Fig. 8 (b). Note the decreasing around time 2800ms in Fig. 8(b), it is due to the MOD_WINOW flow entries in the switches which are still functioning while the burst flows have finished the whole transmission. In another words, this phenomenon is not caused by the packet loss.

To evaluate the effectiveness of SDTCP on switch buffer controller, the buffer sizes are logged during a 14-serconds TCP transmission. The result is shown in the Fig. 10. The SDTCP maintains a stable, low buffer size as the DCTCP while the TCP causes wide oscillations in the buffer size. Although, at the buffer size control aspect, the SDTCP is not performed as well as DCTCP, SDTCP reveal satisfactory performance as we expected.

B. SDTCP performance without background flow

In the previous experiments, SDTCP guarantees the goodput by punishing the long-lived background flows. Indeed, as describe in the abstract, the existence of background flows is an advantage for SDTCP which is design for this scenario. Still, it is possible that all of them are burst flows when the TCP incast happens. In this experiment, we test SDTCP with such setting.

Under the same setup, R1 to S1-S10 as Burst flow1, R2 to S11-S20 as burst flow 2 and R3 to S21-S30 as burst flow 3. Fig. 11 details the goodput of the SDTCP when dealing with multi burst flows with no background flow. At the beginning, three burst flows' throughput grows rapidly. In the meantime, the controller pushes the flow entries to reduce the TCP window of one burst flow while the others maintain high goodput. The congestion notification is triggered for the three times through all. As we can observe in the Fig. 11 (b), the total goodput drop to a low level around 400ms when all of the flows are punished. We describe this scenario in III.D which all the flows are greedy flows and have been inhibited by decreasing the TCP window. Then, the TCP windows of the flows are set to an appropriate value which they can share the goodput equally

Through this experiment, SDTCP maintains 0.9Gbps of goodput, thus demonstrating the advantage of SDTCP when dealing with multi burst flows.

C. Packet_In Packets Pressure test

In the OpenFlow network, the first packet of a flow needs to send to the controller through Packet_In to determine the routing information, and, in the SDTCP, the global information database needs to be built on the controller at the same time. Hence, with large quantities of Packet_In messages springing to the controller, the query delay of the first packet can be an issue of SDTCP. It is highly possible that multi new flows connections are establishing at the same time in the TCP

incast scenario. In order to prove the feasibility of SDTCP, the controller must withstand this pressure.

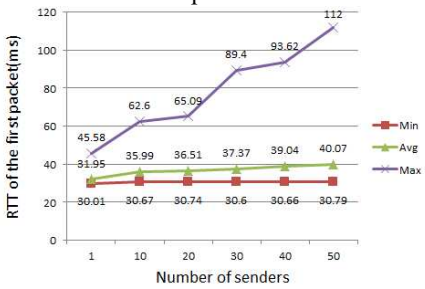


Fig. 12. RTTs with different numbers of senders

The topology is built as the Fig.6 shows and R1 establishes the connections to the senders simultaneously. We record the query delays in 100 trials while the number of sending servers varies.

As shown in the Fig. 12, with the increasing of the senders, the average query delays have no obvious changing. These RTTs contain the 30ms basic transmission delay and the controller handling time. Due to the average RTTs are stable around 40ms, therefore, the high latency of maximum RTTs do not affect the performance of SDTCP seriously. In summary, the result shows that SDTCP can easily handle the TCP incast with 50 senders.

VI. Conclusion And Future Work

In this paper, we present SDTCP, a new transport protocol for providing high-throughput transmission service for the SDN-based data center networks. When burst flows arrive at bottleneck switch and queue length is higher than threshold, SDTCP reduces transmission rate of long-lived flow proactively to guarantee burst flows by adjusting the receive window of ACK packet. SDTCP only needs no modification to existing TCP and makes use of extended OpenFlow, a technology already available in current commodity switches. We evaluate SDTCP via extensive simulations. Our results suggest that SDTCP can make burst flows meet high throughput effectively without starving long-lived flows. Experimental demonstration implementation results show that the SDTCP scheme deals with TCP incast problem excellently as it guarantees the throughput of burst flows and long-lived flows at the same time with no packet loss.

In future work, we are going to use priorities in the controller to determine which flow to be punished. The priorities may composed by 5-tuple (Destination IP, Source IP, Destination Port, Source Port, Protocol), traffic volume, exist time and etc. This approach may enable SDTCP to choose the flow more precisely than just using existing time.

References

[1] D. Nagle, D. Serenyi, and A. Matthews. The Panasas ActiveScale Storage Cluster: Delivering scalable high bandwidth storage. In SC'04: Proceedings of the 2004 ACM/IEEE Conference on Supercomputing, Washington, DC, USA, 2004.

[2] 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 Proc. SIGCOMM, 2010.

[3] T. Benson, A. Akella, and D. A. Maltz, Network traffic characteristics of data centers in the wild, in Proc. of ACM SIGCOMM Internet Measurement Conference (IMC), Melbourne, Australia, Nov. 2010.

[4] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken, The nature of data center traffic: measurements & analysis, in Proc. of ACM SIGCOMM Internet Measurement Conference (IMC), Chicago, Illinois, USA, Nov. 2009.

[5] 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 Proc. of ACM SIGCOMM, Barcelona, Spain, Aug. 2009.

[6] J. Dean and S. Ghemawat, MapReduce: simplified data processing on large clusters, Communications of the ACM, pp. 107–113, 2008.

[7] Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, Dryad: distributed data-parallel programs from sequential building blocks, ACM IGOPS Operating Systems Review, pp. 59–72, 2007.

[8] Wilson C, Ballani H, Karagiannis T, et al. Better never than late: meeting deadlines in datacenter networks. Proceedings of the ACM SIGCOMM 2011 Conference (SIGCOMM'11), Aug 15-19, 2011, Toronto, Canada. New York, NY, USA: ACM, 2011: 50-61.

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

[10] Software-Defined Networking: The New Norm for Networks, White Paper, Open Networking Foundation (ONF), Apr. 2012. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/white-papers/wp-sdn-newnorm.pdf

[11] Keith Kirkpatrick, Software-Defined Networking, ACM Communication, vol. 56, no.9, pp.16-19, 2013.

[12] S. Azodolmolky, P. Wieder, R. Yahyapour, Cloud computing networking: challenges and opportunities for innovations, IEEE Communications Magazine, vol. 51, no. 7, pp. 54-62, 2013.

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

[14] 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, in USENIX FAST, 2008.

[15] R. S. C. L. Peng Cheng, Fengyuan Ren, Catch the Whole Lot in an Action: Rapid Precise Packet Loss Notification in Data Centers, in NSDI 2014.

[16] M. Ghobadi, S. Yeganeh, and Y. Ganjali, Rethinking end-to-end congestion control in software-defined networks, in Proceedings of the 11th ACM Workshop on Hot Topics in Networks. ACM, 2012, pp. 61–66.

[17] Open vSwitch. [Online]. Available: http://openvswitch.org/

[18] Floodlight. [Online]. Available: http://www.projectfloodlight.org/floodlight/

[19] Mininet. [Online]. Available: http://mininet.org/

[20] DCTCP Patch,http://simula.stanford.edu/~alizade/Site/DCTCP.html.

[21] A. Kuzmanovic, A. Mondal, S. Floyd, and K. Ramakrishnan, Adding Explicit Congestion Notification (ECN) Capability to TCP's SYN/ACK Packets, draft-ietf-tcpm-ecnsyn-03 (work in progress), 2007.

[22] Sally Floyd and Tom Henderson. The NewReno modification to TCP's fast recovery algorithm. RFC 2582, April 1999.