ICTCP: Incast Congestion Control for TCP in Data Center Networks[*]

Haitao Wu[⋆], Zhenqian Feng[⋆†], Chuanxiong Guo[⋆], Yongguang Zhang[⋆]
{hwu, v-zhfe, chguo, ygz}@microsoft.com,
[⋆]Microsoft Research Asia, China
[†]School of computer, National University of Defense Technology, China

ABSTRACT

TCP incast congestion happens in high-bandwidth and low-latency networks, when multiple synchronized servers send data to a same receiver in parallel [15]. For many important data center applications such as MapReduce[5] and Search, this many-to-one traffic pattern is common. Hence TCP incast congestion may severely degrade their performances, e.g., by increasing response time.

In this paper, we study TCP incast in detail by focusing on the relationship among TCP throughput, round trip time (RTT) and receive window. Different from the previous approach to mitigate the impact of incast congestion by a fine grained timeout value, our idea is to design an ICTCP (Incast congestion Control for TCP) scheme at the receiver side. In particular, our method adjusts TCP receive window proactively before packet drops occur. The implementation and experiments in our testbed demonstrate that we achieve almost zero timeout and high goodput for TCP incast.

Categories and Subject Descriptors

C.2.1 [**Network Architecture and Design**]: Network topology, Packet-switching networks

General Terms

Algorithms, Design

Keywords

TCP, Incast congestion, Data center networks

1. INTRODUCTION

As the de facto reliable transport layer protocol, TCP (Transport Control Protocol) has been widely used in the Internet and works well. However, recent studies [13, 15] showed that TCP does not work well for many-to-one traffic pattern on high-bandwidth, low-latency networks, where congestion happens when *many* synchronized servers under a same Gigabit Ethernet switch send data to *one* receiver in parallel. Those connections are called as barrier synchronized since the final performance is determined by the slowest TCP connection that suffers timeout due to packet losses. The performance collapse of those many-to-one TCP connections is called TCP incast congestion.

The traffic and network condition in data center networks create the three preconditions for incast congestion as summarized in [15]. First, data center networks are well structured and layered to achieve *high bandwidth* and *low latency*, and the buffer size of ToR (top-of-rack) Ethernet switches is usually small. Second, recent measurement study showed that *barrier synchronized* many-to-one traffic pattern is common in data center networks [9], mainly caused by MapReduce [5] alike applications in data center. Third, the *transmission data volume* for such traffic pattern is usually *small*.

The root cause of TCP incast collapse is that the highly bursty traffic of multiple TCP connections overflow the Ethernet switch buffer in a short period of time, causing intense packet losses and thus TCP retransmission and timeout. Previous solutions, focused on either reducing the waiting time for packet loss recovery by faster retransmissions [15], or controlling switch buffer occupation to avoid overflow by using ECN and modified TCP at both sender and receiver sides[2].

This paper focuses on avoiding packet losses before incast congestion, which is more appealing than recovering after loss. Of course, recovery schemes can be complementary to congestion avoidance. The smaller change we make to the existing system, the better. To this end, a solution that modifies TCP receiver only is preferred than solutions that require switches and

[*]This work was performed when Zhenqian was an intern at Microsoft Research Asia.

ACM CoNEXT 2010, November 30 – December 3 2010, Philadelphia, USA.

routers support (such as ECN) and modifications at both TCP sender and receiver sides.

Our idea is to perform incast congestion avoidance at receiver side by preventing incast congestion. Receiver side is a natural choice since it knows the throughput of all TCP connections and the available bandwidth. The receiver side can adjust the receive window size of each TCP connection, so the aggregate burstiness of all the synchronized senders are under control. We call our design ICTCP (Incast congestion Control for TCP).

But well controlling the receive window is challenging: The receive window should be small enough to avoid incast congestion, but also large enough for good performance and other non-incast cases. A well performed throttling rate for one incast scenario may not fit well for other scenarios due to the dynamics of the number of connections, traffic volumes, network conditions, etc.

This paper addresses the above challenges by a systematically designed ICTCP. We first perform congestion avoidance at system level. We then use per-flow state to fine-grained tune the receive window of each connection at the receiver side. The technical novelties of this work are as follows: 1) To perform congestion control at receiver side, we use the available bandwidth on the network interface as a quota to coordinate the receive window increase of all incoming connections. 2) Our per flow congestion control is performed independently in slotted time of RTT (Round Trip Time) on each connection, which is also the control latency in its feedback loop. 3) Our receive window adjustment is based on the ratio of difference of measured and expected throughput over expected one. This is to estimate the throughput requirement from sender side and adapt receiver window correspondingly. Besides, we find live RTT is necessary for throughput estimation as we observe that TCP RTT in high-bandwidth low-latency network does increase with throughput, even if link capacity is not reached.

We have developed and implemented ICTCP as a Windows NDIS (Network Driver Interface Specification) filter driver. Our implementation naturally support *virtual machines* which are now widely used in data center. In our implementation, ICTCP as a driver locates at hypervisors below virtual machines. This choice removes the difficulty on obtaining the real available bandwidth after virtual interfaces' multiplexing. It also provides a common waist for various TCP stacks in virtual machines. We have built a testbed with 47 Dell servers and a 48-port Gigabit Ethernet switch. Experiments in our testbed demonstrated the effectiveness of our scheme.

The rest of the paper is organized as follows. Section 2 discusses research background. Section3 describes the design rationale of ICTCP. Section 4 presents ICTCP algorithms. Section 5 shows the implementation of ICTCP as a Windows driver. Sections 6 presents experimental results. Section 7 discusses the extension of ICTCP. Section 8 presents related work. Finally, Section 9 concludes the paper.

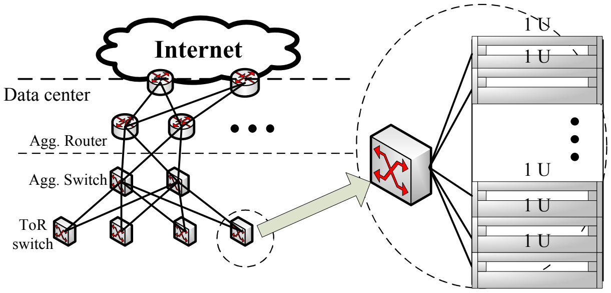

Figure 1: A data center network and a detailed illustration of a ToR (Top of Rack) switch connected to multiple rack-mounted servers

2. BACKGROUND AND MOTIVATION

TCP incast has been identified and described by Nagle et al. [12] in distributed storage clusters. In distributed file systems, files are stored at multiple servers. TCP incast congestion occurs when multiple blocks of a file are fetched from multiple servers. Several application specific solutions have been proposed in the context of parallel file system. With recent progresses on data center networking, TCP incast problem in data center networks has become a practical issue. Since there are various data center applications, a transport layer solution can free application from building their own solutions and is therefore preferred.

In this Section, we first briefly introduce the TCP incast problem, we then illustrate our observations for TCP characteristics on high-bandwidth, low-latency network, and also the root cause of packet loss of incast congestion. After observing TCP receive window is a right controller to avoid congestion, we seek for a general TCP receive window adjustment algorithm.

2.1 TCP incast congestion

In Figure 1, we show a typical data center network structure. There are three layers of switches/routers: ToR (Top of Rack) switch, Aggregate switch and Aggregate router. We also show a detailed case for a ToR connected to dozens of servers. In a typical setup, the number of servers under the same ToR is from 44 to 48, and the ToR switch is a 48-port Gigabit switch with one or multiple 10 Gigabit uplinks.

Incast congestion happens when multiple sending servers under the same ToR switch send data to one receiver server simultaneously. The amount of data transmitted by each connection is relatively small, e.g, 64kbytes. In Figure 2, we show the goodput achieved on those multiple connections versus the number sending servers. Note that we use term *goodput* as it is obtained and observed at application layer. The results are mea-

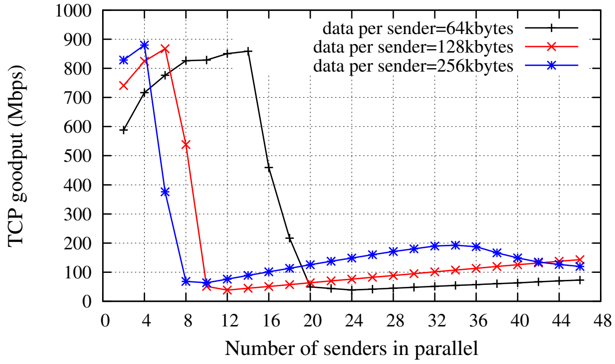

Figure 2: The total goodput of multiple barrier synchronized TCP connections versus the number of senders, where the data traffic volume per sender is a fixed amount

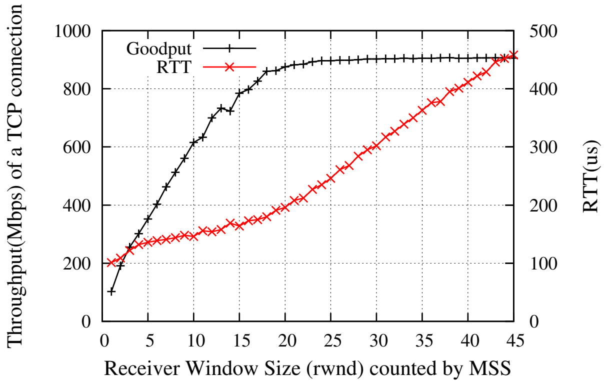

Figure 3: The goodput and RTT of one TCP connection over Gigabit Ethernet versus the receive window size

sured on a testbed with 47 Dell servers connected to one Quanta LB4G 48-port Gigabit switch. Those multiple connections are barrier synchronized. We observe similar goodput trends for three different traffic amounts per server, but with slightly different transition points. Note that in our setup each connection has a same traffic amount with the number of senders increasing, which is used in [13]. [15] uses another setup that the total traffic amount of all senders is a fixed one. Here we just illustrate the problem and will show the results for both setups in Section 6.

TCP throughput is severely degraded on incast congestion, since one ore more TCP connections experience timeout caused by packet drops. TCP incast scenario is common for data center applications. For example, for search indexing we need to count the frequency of a specific word in multiple documents. This job is distributed to multiple servers and each server is responsible for some documents on its local disk. Only after all servers return their counters to the receiving server, the final result can be generated.

2.2 TCP goodput, receive window and RTT

TCP receive window is introduced for TCP flow control, i.e., preventing faster sender from overflowing slow receiver's buffer. The receive window size determines the maximal number of bytes that the sender can transmit without receiving receiver's ACK. Previous study [10] mentioned that a small static TCP receive buffer may throttle TCP throughput and thus prevent TCP incast congestion collapse. However, a static buffer can't work for changed number of connections and can't handle the dynamics on applications' requirements.

As TCP receive window has the ability to control TCP throughput and thus prevent TCP incast collapse, we consider how to dynamically adjust it to the proper value. We start from window based congestion control used in TCP. As we know, TCP uses slow start and congestion avoidance to adjust congestion window at sender side. Directly applying such technique to TCP receive window adjustment certainly won't help as it still requires either losses or ECN marks to trigger window decrease, otherwise the window keeps increasing.

Different from TCP's congestion avoidance, TCP Vegas adjusts its window according to changes of RTT. TCP Vegas makes the assumption that TCP RTT is stable before it reaches network available bandwidth. That is to say, the increase of RTT is only caused by packet queueing at bottleneck buffer. TCP Vegas then adjusts window to keep TCP throughput close to the available bandwidth, by keeping the RTT in a reasonable range. Unfortunately, we find that the increase of TCP RTT in high-bandwidth, low-latency network does not follow such assumption.

In Figure 3, we show the throughput and RTT of one TCP connection between two servers under the same ToR Gigabit switch. The connection lasts for 10 seconds. We define the *base RTT* for a connection as the observed RTT when there is no other traffic and TCP receive window is one MSS (Maximum Segment Size). In our testbed, the *base RTT* is around 100 microseconds, which is much smaller than RTT observed in Internet. From Figure 3, we observe that RTT increases as the TCP receive window increases, when throughput is smaller than the available bandwidth (link capacity in this case). Therefore, in data center network, even if there is no cross traffic, an increase on RTT can't be regarded as a signal for TCP throughput reaching available bandwidth. For example, the base RTT is around 100us, and the RTT increases to 460us with maximal throughput at 906Mbps.

Given that the base RTT is near 100us and a full bandwidth of Gigabit Ethernet, the base BDP (Bandwidth Delay Product) without queue is around 12.5k bytes (100us*1Gbps). For an incast scenario, multiple TCP connections share this small pipe, i.e., the base

BDP and queue are shared by those connections. The small base BDP but high bandwidth for multiple TCP connections is the reason that the switch buffer easily goes to overflow. Meanwhile, we also observe that the receive window size should be controlled since the total receive window size of all connections should be no greater than base BDP plus the queue size. Otherwise packets may get dropped.

3. DESIGN RATIONALE

Our goal is to improve TCP performance for incast congestion, instead of introducing a new transport layer protocol. Although we focus on TCP in data center network, we still require no new TCP option or modification to TCP header. This is to keep backward compatibility, and to make our scheme general enough to handle the incast congestion in future high-bandwidth, low-latency network.

Previous work focused on how to reduce the impact of timeout, which is caused by large number of packet losses on incast congestion. We've shown that the basic RTT in data center network is hundreds of microseconds, and the bandwidth is Gigabit and 10 Gigabit in near future. Given such high-bandwidth and low-latency, we focus on how to perform congestion avoidance to prevent switch buffer overflow. Avoiding unnecessary buffer overflow significantly reduces TCP timeouts and saves unnecessary retransmissions.

We focus on the *classical incast* scenario where dozens of servers connected by a Gigabit Ethernet switch. In this scenario, the congestion point happens right before the receiver. This is to say, the switch port in congestion is actually the *last-hop* of all TCP connections at the incast receiver. Recent measurement study[9] showed that this scenario exists in data center networks, and the traffic between servers under the same ToR switch is actually one of the most significant traffic pattern in data center, as locality has been considered in job distribution. Whether incast exists in more advanced data center topology like recent proposals DCell[7], Fat-tree[1] and BCube[6] is not the focus of this paper.

From Figure 3, we observe that TCP receive window can be used to throttle the TCP throughput, which can be leveraged to handle incast congestion although the receive window is originally designed for flow control. In short, our incast quenching scheme is *to design a window based congestion control algorithm at TCP receiver side*, given the incast scenario we have described, and the requirements we made. The benefit of an incast congestion control scheme at receiver is that the receiver knows how much throughput it has achieved and how much available bandwidth left. While the difficulty and also the challenge at receiver side is that an overly controlled window may constrain TCP performance while less controlled window may not prevent incast congestion.

As the base RTT is at hundreds of microseconds in data center [2], our algorithm is restricted to adjust receive window only for TCP flows with RTT less than 2ms. This constraint is designed to focus on low-latency flows. In particular, if a server in a data center communicates with servers within this data center and servers in the Internet simultaneously, our RTT constraint leaves those long RTT (and low throughput) TCP flows untouched. It also implies that some incoming flows may not follow our congestion control. We will show the robustness of our algorithm with background (even UDP) traffic in Section 6.

We summarize the following three observations which form the base for ICTCP.

First, the available bandwidth at receiver side is the signal for receiver to do congestion control. As incast congestion happens at the last-hop, the incast receiver should detect such receiving throughput burstiness and control the throughput to avoid potential incast congestion. If the TCP receiver needs to increase the TCP receive window, it should also predict whether there is enough available bandwidth to support the increase. Furthermore, the receive window increase of all connections should be jointly considered.

Second, the frequency of receive window based congestion control should be made according to the per-flow feedback-loop delay independently. In principle, the congestion control dynamics of one TCP connection can be regarded as a control system, where the feedback delay is the RTT of that TCP connection. When the receive window is adjusted, it takes at least one RTT time before the data packets following the newly adjusted receive window arrive. Thus, the control interval should be larger than one RTT time, which changes dynamically according to the queueing delay and also system overhead.

Third, a receive window based scheme should adjust the window according to both link congestion status and also application requirement. The receive window should not restrict TCP throughput when there is available bandwidth, and should throttle TCP throughput before incast congestion happens. Consider a scenario where a TCP receive window is increased to a large value but is not decreased after application requirement is gone, then if the application resumes, congestion may happen with traffic surge on such a large receive window. Therefore, the receiver should differentiate whether a TCP receive window over-satisfies the achieved throughput on a TCP connection, and decrease its receive window if so.

With these three observations, our receive window based incast congestion control intends to set a proper receive window to all TCP connections sharing the same last-hop. Considering that there are many TCP connections sharing the bottlenecked last-hop before incast congestion, we adjust TCP receive window to make those connections share the bandwidth fairly. This is because in data center, those parallel TCP connections may belong to the same job and the last finished one determines the final performance. Note that the fairness controller between TCP flows is independent of receive window adjustment for incast congestion avoidance, so that any other fairness category can be deployed if needed.

4. ICTCP ALGORITHM

ICTCP provides a receive window based congestion control algorithm for TCP at end-system. The receive window of all low-RTT TCP connections are jointly adjusted to control throughput on incast congestion. Our ICTCP algorithm closely follow the design points made in Section 3. In this Section, we describe how to set the receive window of a TCP connection, and we discuss how to implement our algorithm at next Section.

4.1 Control trigger: available bandwidth

Without loss of generality, we assume there is one interface on a receiver server, and define symbols corresponding to that interface. Our algorithm can be applied for the scenario that the receiver has multiple interfaces, while the connections on each interface should perform our algorithm independently.

Assume the link capacity of the interface on receiver server is C. Define the bandwidth of total incoming traffic observed on that interface as BW_T, which includes all types of packets, i.e., broadcast, multicast, unicast of UDP or TCP, etc. Then we define the available bandwidth BW_A on that interface as,

$$BW_A = max(0, \alpha * C - BW_T) \qquad (1)$$

where $\alpha \in [0, 1]$ is a parameter to absorb potential oversubscribed bandwidth during window adjustment. In all our implementation and experiment, we have a fixed setting with $\alpha = 0.9$.

In ICTCP, we use available bandwidth BW_A as the quota of all incoming connections to increase receive window for higher throughput. Each flow should estimate the potential throughput increase before its receive window should be increased. Only when there is enough quota (BW_A), the receive window can be increased, and the corresponding quota is consumed to prevent bandwidth oversubscription.

To estimate the available bandwidth on the interface and provide a quota for later receive window increase, we divide time into slots. Each slot consists of two sub-slots with the same length T. *For each network interface, we measure all the traffic received in the first sub-slot, and use it to calculate the available bandwidth as quota for window increase on the second sub-slot.*

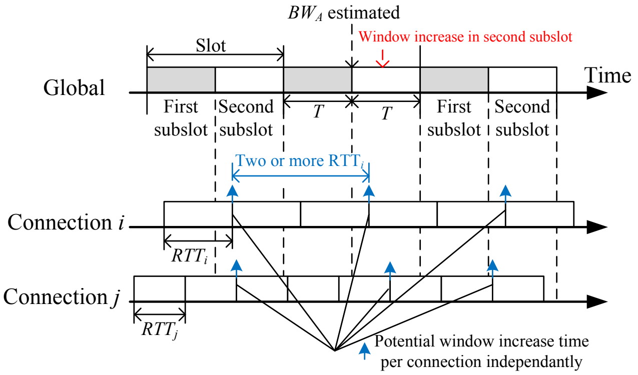

Figure 4: slotted time on global (all connections on that interface) and two arbitrary TCP connections i/j are independent

In Figure 4, the arrowed line marked as "Global" denotes the slot allocation for available bandwidth estimation on a network interface. The first sub-slot is marked in gray color. In the time of first sub-slot, all connections' receive window can't be increased (but can be decreased if needed). The second sub-slot is marked in white in Figure 4. In the second sub-slot, the receive window of any TCP connection can be increased, but the total estimated increased throughput of all connections in the second sub-slot must be less than the available bandwidth observed in the first sub-slot. Note that a decrease of any receive window does not increase the quota, as the quota will only be reset by incoming traffic in the next first sub-slot. We discuss how to choose T and its relationship with per flow control interval in the next subsection.

4.2 Per connection control interval: 2*RTT

In ICTCP, each connection only adjusts its receive window when there is an ACK sending out on that connection. No additional TCP ACK packets are generated only for window adjustment, so that no traffic is wasted. For a TCP connection, after an ACK is sent out, the data packet corresponding to that ACK arrives one RTT later. As a control system, the latency on the feedback-loop is one RTT time of each TCP connection.

Meanwhile, to estimate the throughput of a TCP connection for receive window adjustment, the shortest time scale is an RTT for that connection. Therefore, the control interval for a TCP connection is 2*RTT in ICTCP, as we need one RTT latency for that adjusted window to take effect, and one additional RTT to measure the achieved throughput with that newly adjusted window. Note that the window adjustment interval is performed per connection. We use Connection i and j to represent two arbitrary TCP connections in Figure 4, to show that one connection's receive windows adjustment is independent with the other.

The relationship of sub-slot length T and any flow's control interval is as follows. Since the major purpose of available bandwidth estimation on the first sub-slot is to provide a quota for window adjustment on the second sub-slot, the length T should be determined by the control intervals of all connections. We use a weighted averaged RTT of all TCP connections as T, i.e., $T = \sum_i w_i RTT_i$. The weight w_i is the normalized traffic volume of connection i over all traffic.

In Figure 4, we illustrate the relationship of two arbitrary TCP connections i/j with $RTT_{i/j}$ and the system estimation sub-interval T. Each connection adjust its receive window based on its observed RTT. The time for a connection to increase its receive window is marked with an up arrow in Figure 4. For any TCP connection, if now time is in the second global sub-slot and it observes that the past time is larger than 2*RTT since its last receive window adjustment, it may increase its window based on newly observed TCP throughput and current available bandwidth. Note the RTT of each TCP connection is drawn as a fixed interval in Figure 4. This is just for illustration. We discuss how to obtain accurate and live RTT at receiver side in Section 5.

4.3 Window adjustment on single connection

For any ICTCP connection, the receive window is adjusted based on its incoming *measured throughput* (denoted as b^m) and its *expected throughput* (denoted as b^e). The measured throughput represents the achieved throughput on a TCP connection, also implies the current requirement of the application over that TCP connection. The expected throughput represents our expectation of the throughput on that TCP connection if the throughput is only constrained by receive window.

Our idea on receive window adjustment is to increase window when the difference ratio of measured and expected throughput is small, while decrease window when the difference ratio is large. Similar concept is introduced in TCP Vegas [3] before but it uses throughput difference instead of difference ratio, and it's designed for congestion window at sender side to pursue available bandwidth. ICTCP window adjustment is to set the receive window of a TCP connection to a value that represents its current application's requirement. Oversized receive window is a hidden problem as the throughput of that connection may reach the expected one at any time, and the traffic surge may overflow the switch buffer, which is hard to predict and avoid.

The measured throughput b_i^m is obtained and updated for every RTT on connection i. For every RTT on connection i, we obtain a sample of current throughput, denoted as b_i^s, calculated as total number of received bytes divided by the time interval RTT_i. We smooth measured throughput using exponential filter as,

$$b_{i,new}^m = max(b_i^s, \beta * b_{i,old}^m + (1-\beta) * b_i^s) \qquad (2)$$

Note that the max procedure here is to make b_i^m updated quickly if receive window is increased, especially when window is doubled. The expected throughput of connection i is obtained as,

$$b_i^e = max(b_i^m, rwnd_i/RTT_i) \qquad (3)$$

where $rwnd_i$ and RTT_i are the receive window and RTT for connection i respectively. We have the max procedure to ensure $b_i^m \le b_i^e$.

We define the ratio of throughput difference d_i^b as the ratio of throughput difference of measured and expected throughput over the expected one for connection i.

$$d_i^b = (b_i^e - b_i^m)/b_i^e \qquad (4)$$

By definition, we have $b_i^m \le b_i^e$, thus $d_i^b \in [0,1]$.

We have two thresholds γ_1 and γ_2 ($\gamma_2 > \gamma_1$) to differentiate three cases for receive window adjustment:

1) $d_i^b \le \gamma_1$ or $d_i^b \le MSS_i/rwnd_i$[1], increase receive window if it's now in global second sub-slot and there is enough quota of available bandwidth on the network interface. Decrease the quota correspondingly if the receive window is increased.

2) $d_i^b > \gamma_2$, decrease receive window by one MSS[2] if this condition holds for three continuous RTT. The minimal receive window is 2*MSS.

3) Otherwise, keep current receive window.

In all of our experiments, we have $\gamma_1 = 0.1$ and $\gamma_2 = 0.5$. Similar to TCP's congestion window increase at sender, the increase of the receive window on any ICTCP connection consists of two phases: *slow start* and *congestion avoidance*. If there is enough quota, in slow start phase, the receive window is doubled, while it is enlarged by at most one MSS in congestion avoidance phase. A newly established or long time idle connection is initiated in slow start phase, whenever the above second and third case is met, or the first case is met but there is not enough quota on receiver side, the connection goes into congestion avoidance phase.

4.4 Fairness controller for multiple connections

When the receiver detects that the available bandwidth BW_A becomes smaller than a threshold, ICTCP starts to decrease the receiver window of some selected connections to prevent congestion. Considering that multiple active TCP connections at the same time typically works for the same job in data center, we seek for a method that can achieve fair sharing for all connections without sacrificing throughput. Note that ICTCP does not adjust receive window for flows with RTT larger

[1] This means the throughput difference is less than one MSS with current receive window. It's designed to speed up increase step when receive window is relatively small.

[2,3] The decrease of only one MSS is to follow TCP's design that the right-edge of sender's buffer never left shifted when receive window shrinks. Note we decrease receive window by an outgoing ACK that acknowledges an MSS data.

than 2ms, so fairness is only considered among low-latency flows.

In our experiment, we decrease window for fairness when $BW_A < 0.2C$. This condition is designed for high bandwidth network, where link capacity is under-utilized for most of time. If there is still enough available bandwidth, the requirement of better fairness is not strong as potential impact on achieved throughput.

We adjust receive window to achieve fairness for incoming TCP connections with low-latency on two folds: 1) For window decrease, we cut the receive window by one MSS[3], for some selected TCP connections. We select those connections that have receive window larger than the average window value of all connections. 2) For window increase, this is automatically achieved by our window adjustment described in Section 4.3, as the receive window is only increased by one MSS on congestion avoidance.

5. IMPLEMENTATION

5.1 Software stack

Ideally, ICTCP should be integrated into existing TCP stack, as it is an approach for TCP receive window optimization to achieve better performance.

Although the implementation of ICTCP in TCP stack is natural, this paper chooses a different approach – develop ICTCP as a NDIS driver on Windows OS. The software stack is shown in Figure 5. The NDIS ICTCP driver is implemented in Windows kernel, between TCP/IP and NIC (Network Interface Card) driver, known as a Windows filter driver. Our NDIS driver intercepts TCP packets and modifies the receive window size if needed. The final solution for ICTCP should be in TCP module, while our implementation of ICTCP in driver is to demonstrate its feasibility and performance.

There are several benefits that can be achieved when ICTCP is implemented in a driver: 1) It naturally supports the case for virtual machine, which is widely used in data center. We discuss this point in detail in next subsection. 2) ICTCP needs the incoming throughput in very short time scale (comparable to RTT at hundreds of microseconds) to estimate available bandwidth, and at a driver these information can be easily obtained. Note that the incoming traffic include all types of traffic arrived on that interface, besides TCP. 3) It does not touch TCP/IP implementation in Windows kernel. As a quick and dirty solution, it supports all OS versions, instead of patching one by one to get deployed in a large data center network with various TCP implementations.

As shown in Figure 5, ICTCP driver implementation contains 3 software modules: packet header parser/modifier, flow table and ICTCP algorithm. The parser/modifer implements the functions both to check packet header and modify receive window on TCP header. Flow table maintains the key data structure in the ICTCP driver. A flow is identified by a 5-tuple: source /destination IP address, source/destination port and protocol. The flow table stores per flow information for all the active flows. For a TCP flow, its entry is removed from the table if a FIN/RST packet is observed, or no packets parsed for 15 minutes. The algorithm part implements all the algorithms described in Section 4.

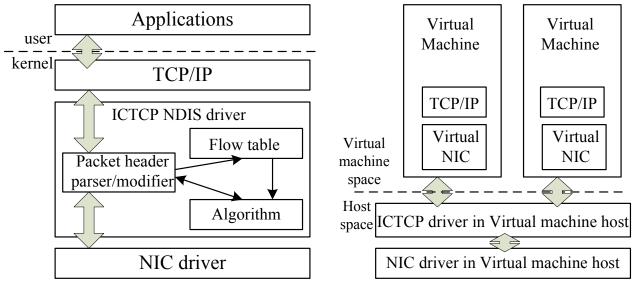

Figure 5: Modules in ICTCP driver and software stack for virtual machine support

The work flow of an ICTCP driver is as follows. 1) When the intermediate driver captures packets through either NDIS sending or receiving entry, it will redirect the packet to header parser module; 2) Packet header is parsed and corresponding information is updated on flow table; 3) ICTCP algorithm module is responsible for receive window calculation. 4) If a TCP ACK packet is sent out, the header modifier may change the receive window field in TCP header if needed.

Our ICTCP driver does not introduce extra CPU overhead for packet checksum when the receive window is modified, as the checksum calculation is loaded to NIC on hardware, which is a normal setup in data center. Besides, there is no packet reordered in our driver. Although it's not the focus of this paper, we measured the CPU overhead introduced by our filter driver on a Dell server PowerEdge R200, Xeon (4CPU)@2.8GhHz, 8G Memory, and compared with the case that our driver is not installed. The overhead is around 5-6% for 1Gbps throughput, and less than 1% for 100Mbps throughput.

5.2 Support for Virtual Machines

Virtual machine is widely used in data centers. When virtual machine is used, the physical NIC capacity on the host server is shared by multiple virtual NICs in the virtual machines. The link capacity of a virtual NIC has to be configured, and usually as a static value in practice. However, to achieve better performance in multiplexing, the total capacity of virtual NICs is typically configured higher than physical NIC capacity as most virtual machines won't be busy at the same time. Therefore, it brings a challenge to ICTCP in virtual machine, as the observed virtual link capacity and available bandwidth does not represent the real value.

One solution is to change the settings of virtual machine NICs, and make the total capacity of all virtual NICs equal to that physical NIC. An alternative solution is to deploy a ICTCP driver on virtual machine host server. The reason for such deployment is to achieve high performance on virtual machine multiplexing. This is a special design for virtual machine case, and won't get conflict even if ICTCP has already been integrated into TCP in virtual machines. The software stack of ICTCP in virtual machine host is illustrated in the right side of Figure 5, where all connections passing the physical NIC are jointly adjusted.

5.3 Obtain fine-grained RTT at receiver

ICTCP is deployed at the TCP receiver side, and it requires to obtain TCP RTT to adjust receive window. From the discussion in Section 2.2, we need a live RTT as it changes with throughput.

We define the reverse RTT as the RTT after a exponential filter at the TCP receiver side. By definition, the reverse RTT is close to the RTT exponentially filtered at the TCP sender side. The reverse RTT can be obtained if there is data traffic on both directions. Considering the data traffic on reverse direction may not be enough to keep obtaining live reverse RTT, we use TCP timestamp to obtain the RTT on reverse direction.

Unfortunately, the RTT implementation in existing TCP module uses a clock on milliseconds granularity. To obtain an accurate RTT for ICTCP in data center network, the granularity should be at microseconds. Therefore, we modify the timestamp counter into 100ns granularity to obtain live and accurate RTT. Note that this does not introduce extra overhead as such granularity time is available on Windows kernel. We believe similar approach can be taken in other OS. Our change of time granularity on TCP timestamp follows the requirements by RFC1323[8].

6. EXPERIMENTAL RESULTS

We deployed a testbed with 47 servers and one Quanta LB4G 48-port Gigabit Ethernet switch. The topology of our testbed is the same as the one shown in the right side of Figure 1, where 47 servers connects to the 48-port Gigabit Ethernet switch with a Gigabit Ethernet interface respectively. The hardware profile of a server is with 2.2G Intel Xeon CPUs E5520 (two cores), 32G RAM, 1T hard disk, and one Broadcom BCM5709C NetXtreme II Gigabit Ethernet NIC (Network Interface Card). The OS of each server is Windows Server 2008 R2 Enterprise 64-bit version. The CPU, Memory and hard disk were never a bottleneck in any of our experiments. We use iperf to construct the incast scenario where multiple sending servers generate TCP traffic to a receiving server under the same switch. The servers in our testbed have their own background TCP connections for various services, but the background traffic amount is very small compared to our generated traffic. The testbed is in an enterprise network with normal background broadcast traffic.

All comparisons are between a full implementation of ICTCP described in Section 5 and a state-of-the-art TCP New Reno with SACK implementation on Windows Server. The default timeout value of TCP on Windows server is 300 milliseconds (ms). Note that all the TCP stacks are the same in our experiments, as ICTCP is implemented in a filter driver at receiver side.

6.1 Fixed traffic volume per server with the number of senders increasing

The first incast scenario we consider is that a number of senders generate the same amount of TCP traffic to a specific receiver under the same switch. Same as the setup in [13] and [4], we fix the traffic amount generated per sending server.

The TCP connections are barrier synchronized per round, i.e., each round finishes only after all TCP connections in it have finished. The goodput shown is the averaged value of 100 experimental rounds. We observe the incast congestion: with the number of sending servers increasing, the goodput per round actually drops due to TCP timeout on some connections. The smallest number of sending servers to trigger incast congestion varies with the traffic amount generated per server: larger data amount, smaller number of sending servers to trigger incast congestion.

6.1.1 ICTCP with minimal receive window at 2MSS

Under the same setup, the performance of ICTCP is shown in Figure 6. We observe that ICTCP achieves smooth and increasing goodput with the number of sending servers increasing. Larger data amount per sending server results in slightly higher goodput achieved. The averaged goodput of ICTCP shows that incast congestion is effectively throttled. And the goodput of ICTCP with various number of sending servers and traffic amount per sending servers show that our algorithm adapts well to different traffic requirements.

We observe that the goodput of TCP before incast congestion is actually higher than that of ICTCP. For example, TCP achieves 879Mbps while ICTCP achieves 607Mbps with 4 sending servers at 256kbytes per server. There are two reasons: 1) During connection initiation phase (slow-start), ICTCP increases window slower than TCP. Actually, ICTCP increases receive window to double for at least every two RTTs while TCP increase its sending window to double for every RTT. Besides, ICTCP increases receive window by one MSS when available bandwidth is low. 2) The traffic amount per sending server is very small, and thus the time taken in "slow-start" dominates the transmission time

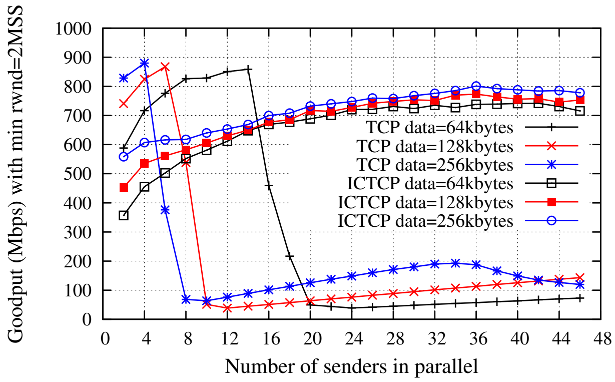

Figure 6: The total goodput of multiple barrier synchronized ICTCP/TCP connections versus the number of senders, where the data traffic volume per sender is a fixed amount

if incast congestion does not happen. Note that low throughput of ICTCP during initiation phase does not affect its throughput during stable phase in larger time scale, e.g., hundreds of milliseconds, which will be evaluated in Section 6.4.

To evaluate the effectiveness of ICTCP on avoiding timeout, we use the ratio of the number of experimental rounds experiencing at least one timeout[4] over the total number of rounds. The ratio of rounds with at least one timeout is shown in Figure 7. We observe that TCP quickly suffer for at least one timeout when incast congestion happens, while the highest ratio for ICTCP experience timeout is 6%. Note that the results in Figure 7 shows that ICTCP is better than DCTCP[2], as DCTCP quickly downgrades to the same as TCP when the number of sending servers is over 35 for static buffer case. The reason for ICTCP to effectively reduce the possibility on timeout is that ICTCP does congestion avoidance and it increases receive window only if there is enough available bandwidth on receiving server. DCTCP relies on ECN to detect congestion, so larger (dynamic) buffer is required to avoid buffer overflow during control latency, i.e., the time before control takes effect.

6.1.2 *ICTCP with minimal receive window at 1MSS*

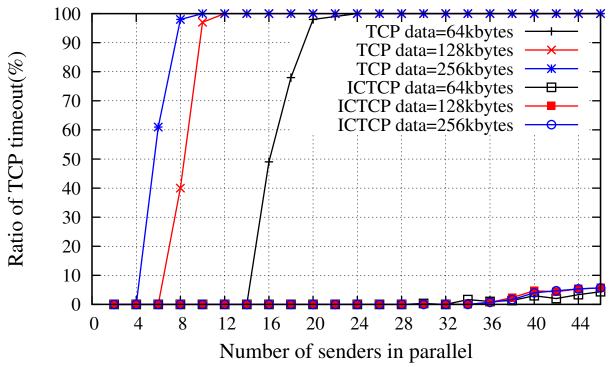

Figure 7: The ratio of experimental rounds that suffer at least one timeout

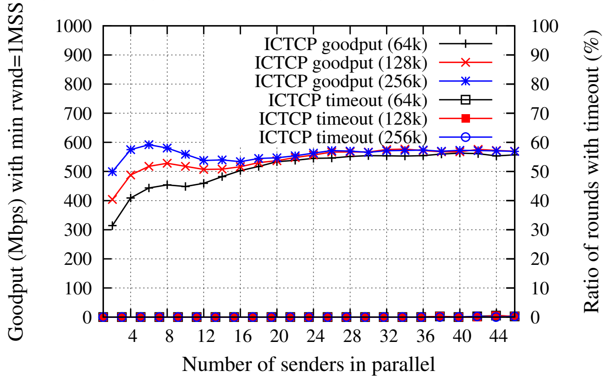

Figure 8: ICTCP goodput and ratio of experimental rounds suffer at least one timeout with minimal receive window as 1MSS

ICTCP has some possibility (although very small) to timeout, since we have a 2MSS minimal receive window. In principle, with the number of connections become larger, the receive window for each connection should become smaller proportionately. This is because the total BDP including the buffer size is actually shared by those connections, and the minimal receive window of those connections determines whether such sharing may cause buffer overflow when the total BDP is not enough to support those connections.

The performance of ICTCP with minimal receive window at 1MSS is shown in Figure 8. We observe that timeout probability is 0, while the averaged throughput is lower than those with 2MSS minimal receive window. For example, for 40 sending servers with 64kbytes per server, the goodput is 741Mbps for 2MSS as shown in Figure 6, while 564Mbps for 1MSS as shown in Figure 8. Therefore, the minimal receive window is a tradeoff between higher averaged incast goodput and lower timeout possibility. Note that the goodput here is only for very short time, 40*64k*8/564Mbps =36ms. For larger request data size and longer connection duration, ICTCP actually achieves goodput closely to link capacity, which is shown in detail in Section 6.4.

6.2 Fixed total traffic volume with the number of senders increasing

The second scenario we consider is with the one discussed in [15, 2], where the total data volume of all servers is fixed and the number of sending servers varies.

We show the goodput and timeout ratio for both TCP and ICTCP under a fixed total traffic amount in Figure

[4] Multiple senders experiencing timeout in the same barrier does not degrade performance proportionately, as the connections are delivered in parallel.

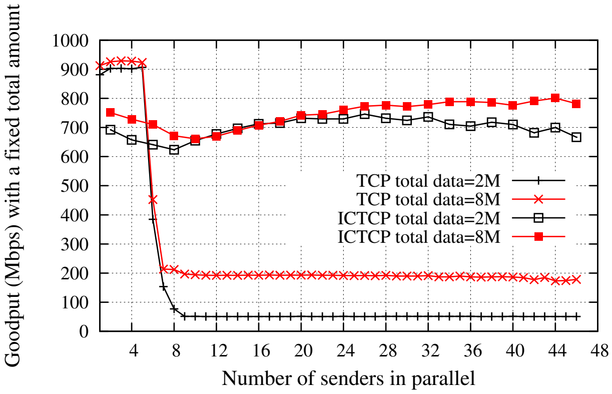

Figure 9: Goodput of ICTCP (with minimal receive window at 2MSS) and TCP under the case that the total data amount from all sending servers is a fixed value

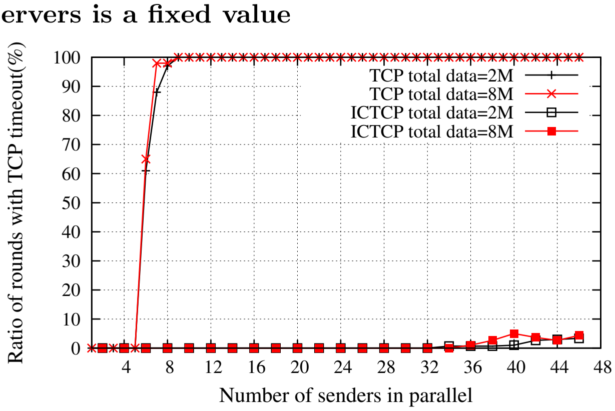

Figure 10: Ratio of timeout for ICTCP (with minimal receive window at 2MSS) and TCP under the case that the total data amount from all sending servers is a fixed value

9 and 10. From Figure 9 we observe that the number of sending servers to trigger incast congestion is close for 2M and 8M bytes total traffic respectively. ICTCP greatly improves the goodput and controls timeout well. Note that we show the case for ICTCP with minimal receive window at 2MSS and skip the case with 1MSS, as the timeout ratio is again 0% for 1MSS.

6.3 Incast with high throughput background traffic

In previous experiments, we do not explicit generate long term background traffic for incast experiments. In the third scenario, we generate a long term TCP connection as background traffic to the same receiving server, and it occupies 900Mbps before incast traffic starts.

The goodput and timeout ratio of TCP and ICTCP are shown in Figure 11 and 12. Compared Figure 11 with Figure 2, the throughput achieved by TCP before incast congestion is slightly lower. ICTCP also achieves slightly lower throughput when the number of sending serves is small. Comparing Figure 12 with Figure 7, the timeout ratio with ICTCP becomes slightly higher when there is a high throughput background connection ongoing. This is because the available bandwidth becomes smaller and thus the initiation of new connections is affected. We also obtain the experimental results for a background UDP connection at 200Mbps, and ICTCP also performs well. We skip the results under background UDP for space limitation.

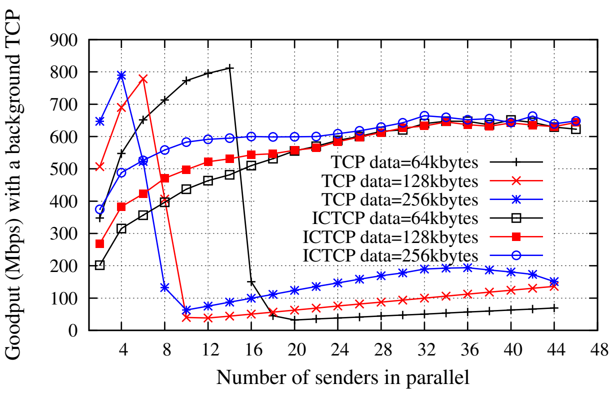

Figure 11: Goodput of ICTCP (with minimal receive window at 2MSS) and TCP under the case with a background long term TCP connection

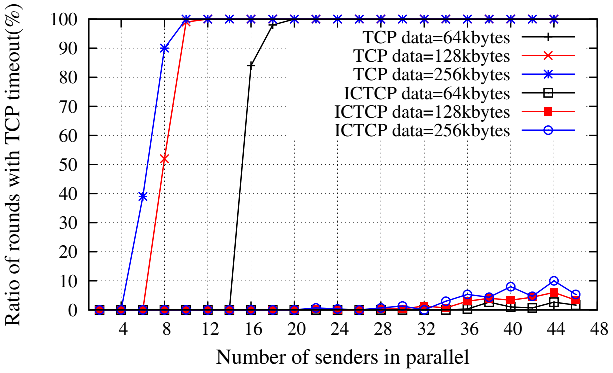

Figure 12: Ratio of timeout for ICTCP (with minimal receive window at 2MSS) and TCP under the case with a background long term TCP connection

6.4 Fairness and long term performance of ICTCP

To evaluate the fairness of ICTCP on multiple connections, we generate 5 ICTCP flows to the same receiving server under the same switch. The flows are started sequentially with 20s interval and 100s duration. The achieved goodput of those 5 ICTCP flows are shown in Figure 13. We observe that the fairness of ICTCP on multiple connections is very good, and the total goodput of multiple connections is close to link capacity at 1Gbps. Note that the goodput here is much larger than that shown in Figure 6, since the traffic volume is much

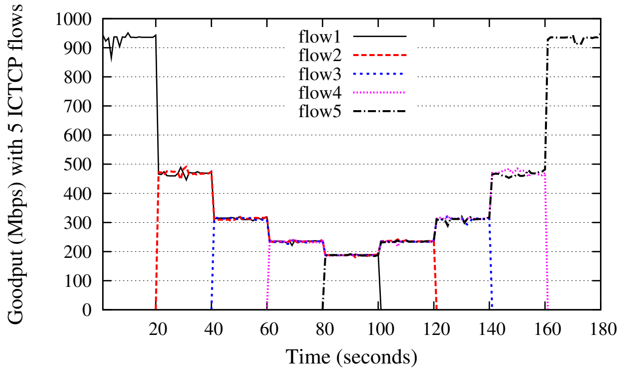

Figure 13: Goodput of 5 ICTCP flows that start sequentially with 20 seconds interval and 100 seconds duration, from 5 sending sever and to the same receiving sever under the same switch

larger and thus slightly longer time cost on slow start phase is not an issue. We also run the same experiments for TCP. TCP achieves same performance before 5 servers and degrades with more connections.

7. DISCUSSIONS

In this section, we discuss three issues related to further extension of ICTCP. The first issue is about the scalability of ICTCP, in particular, how to handle incast congestion with extremely large number of connections. Paper [2] shows that the number of concurrent connections to a receiver on 50ms duration is less than 100 for 90th percentile in a real data center. ICTCP can easily handle the case for 100 concurrent connections with 1MSS as minimal receive window. In principle, if the number of concurrent connections becomes extremely large, then we require to have much smaller minimal receive window to prevent buffer overflow. However, directly using smaller receive window less than MSS may degrade performance greatly. We propose an alternative solution: switching the receive window between several values to achieve effective smaller receive window averaged for multiple RTTs time. For example, 1MSS window for one RTT and 0 window for another RTT could achieve 0.5MSS window in average for 2RTT time. Note that it still needs coordination between multiplexed flows at receiver side to prevent concurrent connections overflow buffer.

The second issue we consider is how to extend ICTCP to handle congestion in general cases where sender and receiver are not under the same switch, and the bottleneck link is not the last-hop to receiver. We consider that ECN can be leveraged to obtain such congestion information. While different from original ECN to only echo congestion signal at receiver side, ICTCP can throttle receive window considering the aggregation of multiple connections.

The third issue is whether ICTCP works for future high-bandwidth low-latency network. We consider a big challenge for ICTCP is that bandwidth may reach 100Gbps, but the RTT may not decrease much. In this case, the BDP is enlarged and the receive window on incast connections also become larger. While in ICTCP, we constrain that only 1MSS reduction is used for window adjustment, which requires longer time to converge if window is larger. To make ICTCP work for 100Gbps or higher bandwidth network, we consider the following solutions: 1) the switch buffer should be enlarged correspondingly. Or 2) the MSS should be enlarged so that window size in number of MSS does not enlarge greatly. This is reasonable as 9kbytes MSS has been available for Gigabit Ethernet.

8. RELATED WORK

TCP has been widely used in Internet and works well for decades. With the advances of new network technologies and emergences of new scenarios, TCP has been improved continuously over the years. In this Section, we first review TCP incast related work, then we discuss previous work using TCP receive window.

Nagle el al. [12] describe TCP incast in the scenarios of distributed storage cluster. TCP incast occurs when a client requests multiple pieces of data blocks from multiple storage servers in parallel. In [10], several application-level approaches have been discussed. Among those approaches, they mentioned that a smaller TCP receiver buffer can be used to throttle data transfer, which is also suggested in [12]. Unfortunately, this type of static setting on TCP receiver buffer size is at application-level and faces the same challenge on how to set the receiver buffer to an appropriate value for general cases.

TCP incast congestion in data center networks has become a practical issue [15]. Since a data center needs to support a large number of applications, a solution at transport layer is preferred. In [13], several approaches at TCP-level have been discussed, focusing on TCP timeout which dominates the performance degradation. They show that several techniques including, alternative TCP implementations like SACK, reduced duplicate ACK threshold, and disabling TCP slow-start can't eliminate TCP incast congestion collapse. Paper [15] presents a practical and effective solution to reduce the impact caused by incast congestion. Their method is to enable a microsecond-granularity TCP timeouts. Faster retransmission is reasonable since the base TCP RTT in data center network is only hundreds of microseconds (Figure 3). Compared with existing widely used milliseconds-granularity TCP timeouts, the interaction of fine granularity timeout and other TCP schemes are still under investigation [4]. 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.

Motivated by the interaction between short flows that require short-latency and long background throughput-oriented flows, DCTCP [2] focuses on reducing the Ethernet switch buffer occupation. 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. Their experimental results show that DCTCP outperforms TCP for TCP incast, but eventually converges to equivalent performance as incast degree increases, e.g., over 35 senders. The difference between ICTCP and DCTCP is that ICTCP only touches TCP receiver, and ICTCP uses the throughput increase estimation to predict whether available bandwidth is enough for receive window increase, and avoids congestion effectively.

To avoid congestion, TCP Vegas[3] has been proposed. In TCP Vegas, sender adjusts congestion window based on the difference of expected throughput and actual throughput. However, there are several issues to apply TCP Vegas directly to a receiver window based congestion control in high-bandwidth low-latency network: 1) As we have shown in Figure 3, the RTT is only hundreds of microseconds, and RTT increases before available bandwidth is reached. The window adjustment in Vegas is based on a stable base RTT, which makes Vegas may over-estimate throughput. 2) In Vegas, the absolute difference of expected and actual throughput is used. While it changes greatly, as queueing latency (also at hundreds of microseconds scale) greatly affects the sample of RTT. This makes the absolute difference defined in Vegas hard to use for window adjustment, and also this is the reason we use the ratio of throughput difference over expected throughput.

TCP receiver buffer [14], receive window and also delayed ACK [11] are used to control the bandwidth sharing between multiple TCP flows at receive side. In the previous work, they focus on the ratio of achieved bandwidth of those multiple independent TCP flows. While our focus is congestion avoidance to prevent packet loss in incast, which is not considered in previous work. Meanwhile, ICTCP adjusts receive window for better fairness among multiple TCP flows only when available bandwidth is small.

9. CONCLUSIONS

In this paper, we have presented the design, implementation, and evaluation of ICTCP, to improve TCP performance for TCP incast in data center networks. Different from previous approaches by using a fine tuned timer for faster retransmission, we focus on receiver based congestion control algorithm to prevent packet loss. ICTCP adaptively adjusts TCP receive window based on the ratio of difference of achieved and expected per connection throughputs over expected ones, as well as the last-hop available bandwidth to the receiver.

We have developed a light-weighted, high performance Window NDIS filter driver to implement ICTCP. Compared with directly implementing ICTCP as part of the TCP stack, our driver implementation can directly support virtual machines, which are becoming prevail in data centers. We have built a testbed with 47 servers together with a 48-port Ethernet Gigabit switch. Our experimental results demonstrate that ICTCP is effective to avoid congestion by achieving almost zero timeout for TCP incast, and it provides high performance and fairness among competing flows.

10. REFERENCES

[1] M. Al-Fares, A. Loukissas, and A. Vahdat. A Scalable, Commodity Data Center Network Architecture. In *Proc. SIGCOMM*, 2008.

[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] L. Brakmo and L. Peterson. TCP Vegas: End to End Congestion Avoidance on a Global Internet. *JSAC*, 1995.

[4] Y. Chen, R. Griffith, J. Liu, R. Katz, and A. Joseph. Understanding TCP Incast Throughput Collapse in Datacenter Networks. In *Proc. WREN*, 2009.

[5] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI'04*, 2004.

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

[7] C. Guo, H. Wu, K. Tan, L. Shi, Y. Zhang, and S. Lu. DCell: A Scalable and Fault Tolerant Network Structure for Data Centers. In *Proc. SIGCOMM*, 2008.

[8] V. Jacobson, R. Braden, and D. Borman. TCP Extensions for High Performance. *RFC1323*, May 1992.

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

[10] E. Krevat, V. Vasudevan, A. Phanishayee, D. Andersen, G. Ganger, G. Gibson, and S. Seshan. On Application-level Approaches to Avoiding TCP Throughput Collapse in Cluster-based Storage Systems. In *Proc. Supercomputing*, 2007.

[11] P. Mehra, A. Zakhor, and C. Vleeschouwer. Receiver-driven bandwidth sharing for TCP. In *Proc. INFOCOM*, 2003.

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

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

[14] N. Spring, M. Chesire, M. Berryman, and V. Sahasranaman. Receiver Based Management of Low Bandwidth Access Links. In *Proc. INFOCOM*, 2000.

[15] 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.