ICTCP: Incast Congestion Control for TCP in Data-Center Networks

Haitao Wu, *Member, IEEE*, Zhenqian Feng, Chuanxiong Guo, *Member, IEEE*, and Yongguang Zhang, *Senior Member, IEEE*

***Abstract*—Transport Control Protocol (TCP) incast congestion happens in high-bandwidth and low-latency networks when multiple synchronized servers send data to the same receiver in parallel. For many important data-center applications such as MapReduce 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 relationships between TCP throughput, round-trip time (RTT), and receive window. Unlike previous approaches, which mitigate the impact of TCP incast congestion by using a fine-grained timeout value, our idea is to design an Incast congestion Control for TCP (ICTCP) scheme on the receiver side. In particular, our method adjusts the TCP receive window proactively before packet loss occurs. The implementation and experiments in our testbed demonstrate that we achieve almost zero timeouts and high goodput for TCP incast.**

***Index Terms*—Data-center networks, incast congestion, TCP.**

I. Introduction

As THE *de facto* reliable transport-layer protocol, Transport Control Protocol (TCP) is widely used on the Internet and generally works well. However, recent studies [1], [2] have shown that TCP does not work well for many-to-one traffic patterns on high-bandwidth, low-latency networks. Congestion occurs when *many* synchronized servers under the same Gigabit Ethernet switch simultaneously send data to *one* receiver in parallel. Only after all connections have finished the data transmission can the next round be issued. Thus, these connections are also called *barrier-synchronized*. The final performance is determined by the slowest TCP connection, which may suffer from timeout due to packet loss. The performance collapse of these many-to-one TCP connections is called TCP incast congestion.

The traffic and network conditions in data-center networks create the three preconditions for incast congestion as summarized in [2]. First, data-center networks are well structured and layered to achieve *high bandwidth* and *low latency*, and the buffer size of top-of-rack (ToR) Ethernet switches is usually small. Second, a recent measurement study showed that a *barrier-synchronized* many-to-one traffic pattern is common in data-center networks [3], mainly caused by MapReduce [4] and similar applications in data centers. Third, the *transmission data volume* for such traffic patterns is usually *small*, e.g., ranging from several hundred kilobytes to several megabytes in total.

The root cause of TCP incast collapse is that the highly bursty traffic of multiple TCP connections overflows the Ethernet switch buffer in a short period of time, causing intense packet loss and thus TCP retransmission and timeouts. Previous solutions focused on either reducing the wait time for packet loss recovery with faster retransmissions [2], or controlling switch buffer occupation to avoid overflow by using ECN and modified TCP on both the sender and receiver sides [5].

This paper focuses on avoiding packet loss before incast congestion, which is more appealing than recovery after loss. Of course, recovery schemes can be complementary to congestion avoidance. The smaller the change we make to the existing system, the better. To this end, a solution that modifies only the TCP receiver is preferred over solutions that require switch and router support (such as ECN) and modifications on both the TCP sender and receiver sides.

Our idea is to perform incast congestion avoidance at the receiver side by preventing incast congestion. The 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 kept under control. We call our design Incast congestion Control for TCP (ICTCP).

However, adequately 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 nonincast cases. A well-performing throttling rate for one incast scenario may not be a good fit for other scenarios due to the dynamics of the number of connections, traffic volume, network conditions, etc.

This paper addresses the above challenges with a systematically designed ICTCP. We first perform congestion avoidance at the system level. We then use the per-flow state to finely tune the receive window of each connection on the receiver side. The technical novelties of this work are as follows: 1) To perform congestion control on the 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 of the slotted time of the round-trip time (RTT) of each connection, which is also the control latency in its feedback loop.

Manuscript received March 29, 2011; revised September 11, 2011; February 15, 2012; and April 09, 2012; accepted April 20, 2012; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor C. Dovrolis. Date of publication May 21, 2012; date of current version April 12, 2013.

H. Wu, C. Guo, and Y. Zhang are with the Wireless and Networking Group, Microsoft Research Asia (MSRA), Beijing 100080, China (e-mail:hwu@microsoft.com; chguo@microsoft.com; ygz@microsoft.com).

Z. Feng was with Wireless and Networking Group, Microsoft Research Asia (MSRA), Beijing 100080, China. He is now with the School of Computer, National University of Defense Technology, Changsha 410073, China.

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TNET.2012.2197411

3) Our receive window adjustment is based on the ratio of the difference between the measured and expected throughput over the expected. This allows us to estimate the throughput requirements from the sender side and adapt the receiver window accordingly. We also find that live RTT is necessary for throughput estimation as we have observed that TCP RTT in a high-bandwidth low-latency network increases with throughput, even if link capacity is not reached.

We have developed and implemented ICTCP as a Windows Network Driver Interface Specification (NDIS) filter driver. Our implementation naturally supports *virtual machines* that are now widely used in data centers. In our implementation, ICTCP as a driver is located in hypervisors below virtual machines. This choice removes the difficulty of 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 this paper is organized as follows. Section II discusses research background. Section III describes the design rationale of ICTCP. Section IV presents the ICTCP algorithms. Section VI shows the implementation of ICTCP as a Windows driver. Section VII presents experimental results. Section VIII discusses the extension of ICTCP. Section IX presents related work. Finally, Section X concludes the paper.

II. Background and Motivation

TCP incast has been identified and described by Nagle *et al.* [6] in distributed storage clusters. In distributed file systems, the files are deliberately stored in multiple servers. However, TCP incast congestion occurs when multiple blocks of a file are fetched from multiple servers at the same time. Several application-specific solutions have been proposed in the context of parallel file systems. With recent progress in data-center networking, TCP incast problems in data-center networks have become a practical issue. Since there are various data-center applications, a transport-layer solution can obviate the need for applications to build their own solutions and is therefore preferred.

In this section, we first briefly introduce the TCP incast problem, then illustrate our observations for TCP characteristics on high-bandwidth, low-latency networks. Next, we explore the root cause of packet loss in incast congestion, and finally, after observing that the TCP receive window is the right controller to avoid congestion, we seek a general TCP receive window adjustment algorithm.

A. TCP Incast Congestion

In Fig. 1, we show a typical data-center network structure. There are three layers of switches/routers: the ToR switch, the Aggregate switch, and the 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 ranges from 44 to 48, and the ToR switch is a 48-port Gigabit switch with one or multiple 10-Gb uplinks.

Incast congestion happens when multiple sending servers under the same ToR switch send data to one receiver server

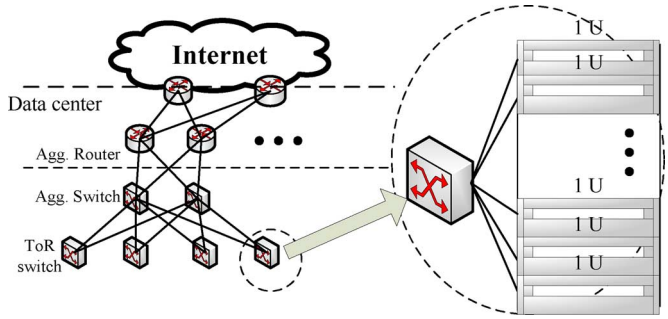


Fig. 1. Data-center network and a detailed illustration of a ToR switch connected to multiple rack-mounted servers.

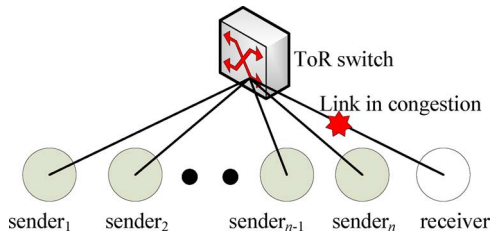


Fig. 2. Scenario of incast congestion in data-center networks, where multiple (n) TCP senders transmit data to the same receiver under the same ToR switch.

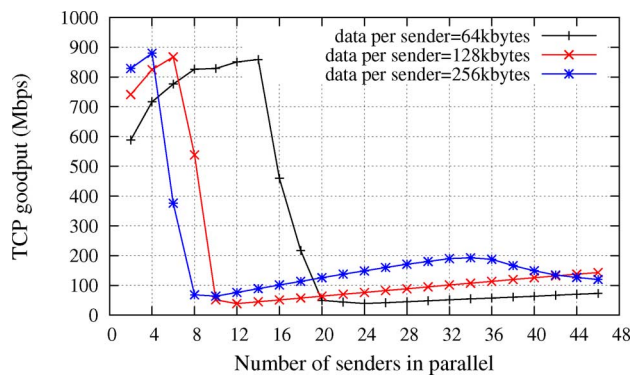


Fig. 3. Total goodput of multiple barrier-synchronized TCP connections versus the number of senders, where the data traffic volume per sender is a fixed amount.

simultaneously, as shown in Fig. 2. The amount of data transmitted by each connection is relatively small, e.g, 64 kB. In Fig. 3, we show the goodput achieved on multiple connections versus the number of sending servers. Note that we use the term *goodput* as it is effective throughput obtained and observed at the application layer. The results are measured on a testbed with 47 Dell servers (at most 46 senders and one receiver) connected to one Quanta LB4G 48-port Gigabit switch. The multiple TCP connections are barrier-synchronized in our experiments. We first establish multiple TCP connections between all senders and the receiver, respectively. Then, the receiver sends out a (very small) request packet to ask each sender to transmit data, respectively, i.e., multiple requests packets are sent using multiple threads. The TCP connections are issued round by round, and one round ends when all connections on that round have finished their data transfer to the receiver.

We observe similar goodput trends for three different traffic amounts per server, but with slightly different transition points.

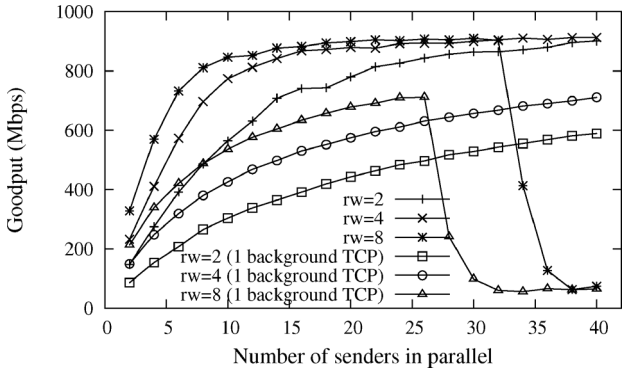


Fig. 4. Incast goodput of capped and fixed receive window (rw) with and without background TCP connection. The value of rw is with MSS, and the background TCP connection is not capped by the receive window.

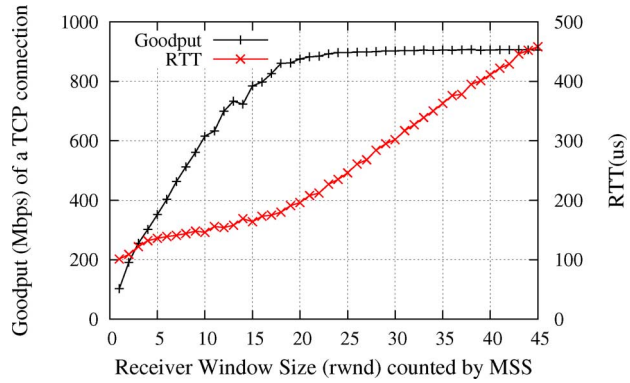


Fig. 5. Goodput and RTT of one TCP connection over Gigabit Ethernet versus the receive window size.

Note that in our setup, each connection has the same traffic amount with the number of senders increasing, which is used in [1]. Reference [2] uses another setup, in which the total traffic amount of all senders is a fixed one, so that the data volume per server per round decreases when the number of senders increases. Here, we just illustrate the incast congestion problem and later will show the results for both setups in Section VII.

TCP throughput is severely degraded by incast congestion since one or more TCP connections can experience timeouts caused by packet drops. TCP variants sometimes improve performance, but cannot prevent incast congestion collapse since most of the timeouts are caused by full window losses [1] due to Ethernet switch buffer overflow. The 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 counts to the receiving server can the final result be generated.

From [3], we know that in a data center, traffic under the same ToR is actually a significant pattern known as *work-seeks-bandwidth*, as locality has been considered during job assignment. Considering that all traffic has to cross the ToR switches before reaching the destination server in the tree-like topology of a data-center network, we focus on the incast congestion on the *last hop*, i.e., output ports of ToR Ethernet switches as shown in Fig. 2.

B. TCP Goodput, Receive Window, and RTT

The TCP receive window is introduced for TCP flow control, i.e., preventing a faster sender from overflowing a slow receiver's buffer. The receive window size determines the maximum number of bytes that the sender can transmit without receiving the receiver's ACK. A previous study [7] mentioned that a small static TCP receive buffer may throttle TCP throughput and thus prevent TCP incast congestion collapse. To demonstrate the effect of a capped receive window, we cap the receive window using varied socket buffer size and show the performance of incast in Fig. 4.

We observe that an optimal receive window exists to achieve high goodput for a given number of senders. As an application-layer solution, a capped and well-tuned receive window with a socket buffer may work for a specific application in a static network (e.g., [6]), but not for a data center with fairly diverse traffic and application requirements. For example, when there is a background TCP connection that is not capped by the receive window, the optimal receive window value for incast application may totally change. The background connection can be generated by other applications, or even from other VMs in the same host server. Thus, *a static buffer* cannot work for a changing number of connections and cannot handle the dynamics of the applications' requirements.

As the 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 with the window-based congestion control used in TCP. As we know, TCP uses slow start and congestion avoidance to adjust the congestion window on the sender side. Directly applying such a technique to the TCP receive window adjustment certainly will not help as it still requires either losses or ECN marks to trigger a window decrease, otherwise the window keeps increasing.

In contrast to TCP's congestion avoidance, TCP Vegas adjusts its window according to the changes in RTT. TCP Vegas makes the assumption that TCP RTT is stable before it reaches the available bandwidth of the network. That is to say, the increase of RTT is only caused by packet queueing at the bottleneck buffer. TCP Vegas then adjusts the 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 networks does not follow such an assumption.

In Fig. 5, we show the throughput and RTT of one TCP connection between two servers under the same ToR Gigabit switch. We fix the receive window by setting the socket buffer: one connection for one receive window, and each connection lasts for 10 s to obtain the throughput and RTT with the receive window. We define the *base RTT* for a connection as the observed RTT when there is no other traffic and the TCP receive window is one maximum segment size (MSS). In our testbed, the *base RTT* is around 100 μs, which is much smaller than RTT observed on the Internet. From Fig. 5, we observe that RTT increases from 100 to 200 μs when the TCP receive window increases from 1 to 20 MSS, while the throughput is smaller than the available bandwidth at around 900 Mb/s. The

increase of RTT before the available bandwidth is reached is caused by queueing at the sender's network adaptor. Therefore, in a data-center network, even if there is no cross traffic, an increase in RTT cannot be regarded as a signal for TCP throughput reaching the available bandwidth. Meanwhile, RTT measurement at such fine timescales is greatly affected by systems, thereby inaccurate values may mislead TCP Vegas's control loop.

C. Reasons for Incast Congestion

Incast congestion happens when the switch buffer overflows because the network pipe is not large enough to contain all TCP packets injected into the network. The capacity of the network pipe is known as bandwidth delay product (BDP). The network delay consists of three parts: 1) the transmission delay and propagation delay, which are relatively constant; 2) the system processing delay at the sender and receiver server, which increases as the window size increases; and 3) the queueing delay at the Ethernet switch. We define the base delay as the RTT observed when the receive window size is one MSS, which covers the first two parts without queueing. Correspondingly, we have a base BDP, which is the network pipe without queueing at the switch. For the incast scenario, the total BDP consists of two parts: the base BDP and the queue size of the output port on the Ethernet switch.

Given that the base RTT is around 100 μs with the full bandwidth of Gigabit Ethernet, the base BDP without queuing is around 12.5 kB (100 μs $\times$ 1 Gb/s). Considering that the Ethernet packet size is 1.5 kB, the base BDP only holds around eight TCP data packets.

The ToR switch is usually low-end (compared to high-layer ones), and thus the queue size is not very large. Taking the Quanta LB4G 48-port Gigabit switch as an example, it has 4 MB cache so that it has 85 kB for each port (if equally divided).[1]

For an incast scenario, multiple TCP connections share this small pipe, i.e., the base BDP plus the queue of a switch port are shared by those connections. The small base BDP but high bandwidth for multiple TCP connections is the reason that the switch buffer easily overflows. In principle, the total number of packets on the flight should be no larger than the BDP, otherwise packets may get dropped at the switch output port.

To constrain the number of packets on the flight, TCP has two windows: a congestion window on the sender side and a receive window on the receiver side. This paper chooses the TCP receive window adjustment as its solution space. If the TCP receive window sizes are properly controlled, the total receive window size of all connections should be no greater than the base BDP plus the queue size, as we will discuss later in Section V. Therefore, no packets are dropped, and incast congestion can be eliminated. In practice, we seek an adaptive scheme to adjust the TCP receive window to avoid incast congestion. We discuss our rationale in Section III.

[1]Advanced buffer maintenance schemes like shadow memory can dynamically allocate more idle buffers to one port temporally, but this may introduce other issues such as the interaction of long-fat connections and short real-time connections, which is addressed in paper [5].

III. Design Rationale

Our goal is to improve TCP performance for incast congestion without introducing a new transport-layer protocol. Although we focus on TCP in data-center networks, we do not require any new TCP options or modifications to the TCP header. Our transport-layer solution keeps backward compatibility on the protocol and programming interface and makes our scheme general enough to handle the incast congestion in future high-bandwidth and low-latency networks. Note that as a transport-layer solution, we make no assumptions about the application requirements. Therefore, the TCP connections could be incast or not, and the coexistence of incast and nonincast connections is achieved.

Previous work focused on how to mitigate the impact of timeouts, which are caused by a large amount of packet loss on incast congestion. We have shown that the base RTT in data-center networks is hundreds of microseconds, and the bandwidth is a gigabit and will be 10 Gb in the 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 *typical incast* scenario where dozens of servers are connected by a Gigabit Ethernet switch. In this scenario, the congestion point happens right before the receiver. That is to say, the switch port in congestion is actually the *last hop* of all TCP connections to the incast receiver. A recent measurement study[3] 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 patterns in data centers, as locality has been considered in job distribution. Whether an incast exists in more advanced data-center topology like recent proposals DCell [8], Fat-tree [9], and BCube [10] is not the focus of this paper.

From Fig. 5, we observe that the TCP receive window can be used to throttle the TCP throughput, as it can be leveraged to handle incast congestion even though the receive window was originally designed for flow control. In short, our incast quenching scheme is *designed as a window-based congestion control algorithm on the TCP receiver side*, given the incast scenario we have described and the requirements we indicated. The benefit of an incast congestion control scheme at the receiver side is that the receiver knows how much throughput it has achieved and how much available bandwidth remains. The difficulty at the receiver side is that an overly throttled window may constrain TCP performance, while an oversized window may not prevent incast congestion.

As the base RTT is hundreds of microseconds in data centers [5], our algorithm is restricted to adjust the receive window only for TCP flows with RTT less than 2 ms. 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 on the Internet simultaneously, our RTT constraint leaves 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 VII.

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

First, the available bandwidth at the receiver side is the signal for the receiver to perform 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 before the data packets following the newly adjusted receive window arrive. Thus, the control interval should be larger than one RTT, which changes dynamically according to queueing delay and system overhead.

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

Based upon these three observations, our receive-window-based incast congestion control is intended to set a proper receive window for 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 the TCP receive window to make those connections share the bandwidth equally. This is because in a data center, parallel TCP connections may belong to the same job, where the last one finished determines the final performance. Note that the fairness controller between TCP flows is independent of the receive window adjustment for incast congestion avoidance, so that any other fairness category such as proportional to weights can be deployed if needed.

IV. ICTCP Algorithm

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

A. Control Trigger: Available Bandwidth

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

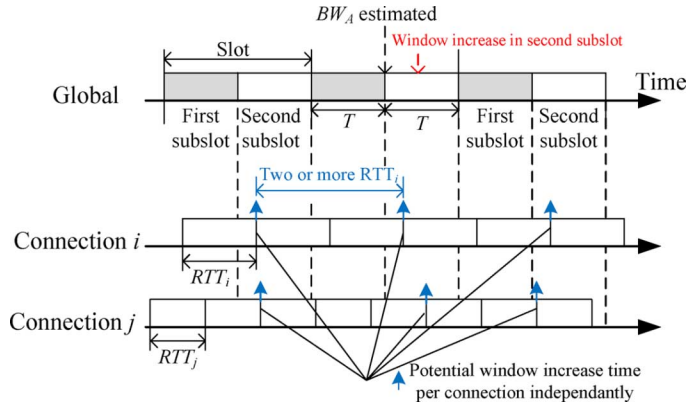


Fig. 6. Slotted time on global (all connections on that interface) and two arbitrary TCP connections i/j are independent.

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

$$\mathrm{BW_A} = \max(0, \alpha * C - \mathrm{BW_T}) \tag{1}$$

where $\alpha \in [0, 1]$ is a parameter to absorb potential oversubscribed bandwidth during window adjustment. A larger α (closer to 1) indicates the need to more conservatively constrain the receive window and higher requirements for the switch buffer to avoid overflow; a lower α indicates the need to more aggressively constrain the receive window, but throughput could be unnecessarily throttled. In all of our implementations and experiments, we have a fixed setting of $\alpha = 0.9$.

In ICTCP, we use available bandwidth $\mathrm{BW_A}$ as the quota for all incoming connections to increase the receive window for higher throughput. Each flow should estimate the potential throughput increase before its receive window is increased. Only when there is enough quota ($\mathrm{BW_A}$) can the receive window 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 a later receive window increase, we divide the time into slots. Each slot consists of two subslots of the same length T. *For each network interface, we measure all the traffic received in the first subslot and use it to calculate the available bandwidth as a quota for window increase on the second subslot.* The receive window of any TCP connection is never increased at the first subslot, but may be decreased when congestion is detected or the receive window is identified as being oversatisfied, which will be discussed in Section IV-C.

In Fig. 6, the arrowed line marked by "Global" denotes the slot allocation for available bandwidth estimation on a network interface. The first subslot is marked in gray. During the first

subslot, none of the connections' receive windows can be increased (but they can be decreased if needed). The second subslot is marked in white in Fig. 6. In the second subslot, the receive window of any TCP connection can be increased, but the total estimated increased throughput of all connections in the second subslot must be less than the available bandwidth observed in the first subslot. 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 subslot. We discuss how to choose T and its relationship to the per-flow control interval next.

B. Per-Connection Control Interval: 2*RTT

In ICTCP, each connection adjusts its receive window only when an ACK is sending out on that connection. No additional pure TCP ACK packets are generated solely for receive 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 for each TCP connection, respectively.

Meanwhile, to estimate the throughput of a TCP connection for a receive window adjustment, the shortest timescale 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 the adjusted window to take effect, and one additional RTT to measure the achieved throughput with the newly adjusted receive window. Note that the window adjustment interval is performed per connection. We use connections i and j to represent two arbitrary TCP connections in Fig. 6 to show that one connection's receive window adjustment is independent from the other.

The relationship of subslot length T and any flow's control interval is as follows: Since the major purpose of available bandwidth estimation on the first subslot is to provide a quota for window adjustment on the second subslot, length T should be determined by the control intervals of all active connections. The changed throughput of any connection is with its RTT, and thus T should be with the RTT to represent the changes in available bandwidth. We use a weighted average RTT of all TCP connections as T, i.e., $T = \sum_i w_i \mathrm{RTT}_i$. The weight w_i is the normalized traffic volume of connection i that has updated RTT_i in the last time period T.

In Fig. 6, we illustrate the relationship of two arbitrary TCP connections i/j with $\mathrm{RTT}_{i/j}$ and the system estimation subinterval T. Each connection adjusts its receive window based on the observed RTT. The time it takes for a connection to increase its receive window is marked with an up arrow in Fig. 6. For TCP connection i, if "now" is in the second global subslot and the elapsed time is larger than $2 * \mathrm{RTT}_i$ since its last receive window adjustment, it may increase its window based on the newly observed TCP throughput and current available bandwidth. Note the RTT of each TCP connection is drawn as a fixed interval in Fig. 6. This is just for illustration. We discuss how to obtain accurate and live RTT on the receiver side in Section VI.

C. 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 and also implies the current requirements of the application over that TCP connection. The expected throughput represents our expectation for the throughput on that TCP connection if the throughput is only constrained by the receive window.

Our idea for receive window adjustment is to increase the receive window when the ratio of the difference between measured and expected throughput over the expected one is small, and to decrease the receive window when the ratio is large. A similar concept has previously been introduced in TCP Vegas [11], but it uses the throughput difference instead of the ratio of throughput difference, and it is designed for the congestion window on the sender side to pursue available bandwidth. ICTCP window adjustment sets the receive window of a TCP connection to a value that represents its current application's requirements. An 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, a situation that is hard to predict and avoid.

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

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

β is the exponential factor, and the default value of β is set to 0.75. Note that the max procedure here is to update b_i^{m} quickly if the receive window is increased, especially when the receive window is doubled. The expected throughput of connection i is obtained as

$$b_i^{\mathrm{e}} = \max(b_i^{\mathrm{m}}, \mathrm{rwnd}_i/\mathrm{RTT}_i) \tag{3}$$

where rwnd_i is the receive window of connection i. We have the max procedure to ensure $b_i^{\mathrm{m}} \leq b_i^{\mathrm{e}}$.

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

$$d_i^b = (b_i^{\mathrm{e}} - b_i^{\mathrm{m}})/b_i^{\mathrm{e}}. \tag{4}$$

By definition, we have $b_i^{\mathrm{m}} \leq b_i^{\mathrm{e}}$, 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) $d_i^b \leq \gamma_1$ or $d_i^b \leq \mathrm{MSS}_i/\mathrm{rwnd}_i$[2]: Increases the receive window if it is in the global second subslot and there is enough quota of available bandwidth on the network interface; decreases the quota correspondingly if the receive window is increased.

[2]This means the throughput difference is less than one MSS with the current receive window. It is designed to speed up the window increasing steps when the receive window is relatively small.

2) $d_i^b > \gamma_2$: Decrease the receive window by 1 MSS[3] if this condition holds for three continuous RTT. The minimal receive window is 2*MSS.
3) Otherwise, keep the current receive window.

The available bandwidth calculated at the end of the first subslot is used for the quota of the second subslot right after the first one. The potential throughput increase of connection i is estimated as the increase in the receive window divided by RTT_i. The quota is consumed by First Come First Service (FIFS), determined by the order of ACKs sent on the second subslot.

In all of our experiments, we had $\gamma_1 = 0.1$ and $\gamma_2 = 0.5$. Note that we set $\gamma_2 = 0.5$ to conservatively decrease the receive window from the ratio of measured and expected throughput obtained in our experiments for greedy connections. Similar to TCP's congestion window increase at the 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 the slow start phase, the receive window is doubled, while it is enlarged by at most one MSS in the congestion avoidance phase. A newly established or prolonged idle connection is initiated in the slow start phase. Whenever the second and third conditions are met, or the first condition is met but there is not enough quota on the receiver side, the connection goes into the congestion avoidance phase.

D. Fairness Controller for Multiple Connections

When the receiver detects that the available bandwidth BW_A has become smaller than the threshold, ICTCP starts to decrease the receiver window of the selected connections to prevent congestion. Considering that multiple active TCP connections typically work on the same job at the same time in a data center, we have sought a method that can achieve fair sharing for all connections without sacrificing throughput. Note that ICTCP does not adjust the receive window for flows with an RTT larger than 2 ms, so fairness is only considered among low-latency flows.

In our experiment, we decrease the receive window for fairness when $\mathrm{BW}_A < 0.2C$. This condition is designed for high-bandwidth networks, where link capacity is underutilized most of the time. If there is still enough available bandwidth, we argue that the requirement of better fairness is not strong considering the potential impact on achieved throughput when decreasing the receive window. The purpose of the $0.2\mathrm{C}$ gap is to leave enough room for other flows to increase their receive window, and should be larger than the increased throughput when the receive window of a flow is increased by 1 MSS.

We adjust the receive window to achieve fairness for incoming TCP connections with low latency as follows: 1) For a window decrease, we cut the receive window by 1 MSS,[3] for some selected TCP connections. We select those connections that have a receive window larger than the average window value of all connections. 2) For a window increase, this is automatically achieved by our window adjustment described in Section IV-C, as the receive window is only increased by 1 MSS during congestion avoidance. In principle, the receive window decrease only happens when the available bandwidth on that interface is small. Furthermore, the connection with the larger receive window is decreased slightly to achieve fairness. If all connections happen to have the same receive window, then none of them decrease the receive window.

Readers may raise some questions concerning our interpretation of fairness to address a congestion control scheme. TCP uses additive increase multiplicative decrease (AIMD) to achieve both stability and fairness among flows sharing the same bottleneck. However, MD happens only when packet loss is observed for a TCP connection. In DCTCP, packet loss during incast congestion due to buffer overflow is largely eliminated. Consider a scenario in which some flows have a large receive window (because they started earlier) while others have a much smaller receive window (because they started later), and the link capacity is almost reached. In this case, none of the connections can increase their receive windows as there is not enough available bandwidth. This situation may persist as long as there is no packet loss so that unfairness between flows becomes an issue. Therefore, we propose slightly reducing the receive window for flows that have a larger receive window, and the throughput of all TCP connections should smoothly converge.

[3]The decrease of only one MSS follows TCP's recommendation that a TCP receiver should not shrink the window, i.e., moves the right window edge to the left [12]. We achieve it by decreasing the receive window with an outgoing ACK that acknowledges one MSS-sized data. Advanced receive-window decreasing schemes may take into account the volume of bytes newly covered by the ACK packet so that a larger receive window decrease by one ACK is possible.

V. Analysis

In this section, we present a simplified flow model for ICTCP in a steady state. In the steady-state control loop, we assume that ICTCP has successfully optimized the receive window of all TCP flows. We are interested in how much buffer space ICTCP needs in the steady state. We use it to judge whether the buffer space requirement of ICTCP is reasonable for existing low-end Ethernet switch buffer space in practice.

We assume that there are n infinitely long-lived TCP flows under the control of ICTCP. All flows go to the same destination server as shown in Fig. 2. The receive window size of these flows is assumed to be $w_i, i = 1, \ldots, n$. The RTT of the flows is denoted as $r_i, i = 1, \ldots, n$. The link capacity is denoted as C, and packet size is denoted as δ.

We assume the base RTT for those flows is the same, and denoted by R. Like the RTT shown in Fig. 5, the base RTT is the least RTT experienced for a connection when there is no queue at the switch. As the queue builds up at the output port of the switch, the RTT of all flows keeps growing. Correspondingly, the base BDP (BDP without queuing) is $R \cdot C$, where C is the bottleneck link capacity.

For connection i, its base BDP is denoted as bdp_i. In the steady state, we assume that all the connections have occupied the base BDP, i.e.,

$$\sum_{i=1}^{n} \mathrm{bdp}_i = R \cdot C. \tag{5}$$

Therefore, in the steady state, the number of data packets for connection i in the switch buffer is bounded by

$$q_i \leq w_i - \mathrm{bdp}_i/\delta. \tag{6}$$

To meet the condition where there are no packet drops for any connection, we have the buffer size Q as

$$Q \geq \sum_{i=1}^{n} q_i \delta. \tag{7}$$

We consider the worst case where a synchronization of all connections happens. In this case, the packets on the flight are all data packets, and there are no ACK packets in the backward direction. For the worst case under this assumption, all those data packets are in the forward direction, in transmission, or waiting in the queue. To ensure there is no packet loss, the queue size Q should be large enough to store packets on the flight for all connections. Hence, in the worst case, if all connections have the same receive window w, we have

$$Q \geq nw\delta - R \cdot C. \tag{8}$$

The synchronization of all connections is likely in an incast scenario, as the senders are transmitting data to the receiver almost at the same time. Note that the synchronization assumption greatly simplifies the receive window adjustment of ICTCP. In practice, packets arrive at the receiver in order, making the connections with earlier packet arrivals have a larger receive window.

We calculate the buffer size requirement to avoid incast buffer overflow from the above equation using an example of a 48-port Ethernet switch. The link capacity C is 1 Gb/s, and the base RTT R is 100 μs, so the BDP $R \cdot C = 12\,500$ B. The default packet length MSS on Ethernet δ is 1500 B. The minimal receive window is by default 2MSS. When considering a 48-port Gigabit Ethernet switch, to avoid packet loss in a synchronized incast scenario with a maximum of 47 senders, the buffer space should be larger than $47 * 2 * 1500 - 12\,500 = 128.5$ kB. Preparing for the worst case, incast congestion may happen at all ports simultaneously, then the total buffer space required is $128.5 * 48 \approx 6$ MB.

We then consider the case of low-end ToR switches, which are also called switches with a shallow buffer in [5]. The switch in our testbed is a Quanta LB4G 48-port Gigabit Ethernet switch, which has a small buffer of about 4 MB. Correspondingly, by a reverse calculation, if the buffer size is 4 MB and the minimal receive window w is 2MSS, then at most 32 senders can be supported. Fortunately, existing commodity switches use dynamic buffer management to allocate a buffer pool shared by all ports. That is to say, if the number of servers is larger than 32, incast congestion may happen with some probability, which is determined by the usage of the shared buffer pool.

VI. Implementation

In this section, we describe the implementation of ICTCP, which is developed in an NDIS driver on the Windows OS. We have implemented the algorithms described in Section IV.

A. Software Stack

Ideally, ICTCP should be integrated into an existing TCP stack, as it is an approach to optimize TCP receive window for better performance. Other receive window optimization schemes, e.g., TCP receive window scaling (using a larger TCP receive window for better performance over a high BDP network), have been implemented in recent Windows OSs (Vista and later) and can be configured by command. We believe that the ICTCP approach on receive window adaptations can be implemented similarly for high-bandwidth, low-latency networks.

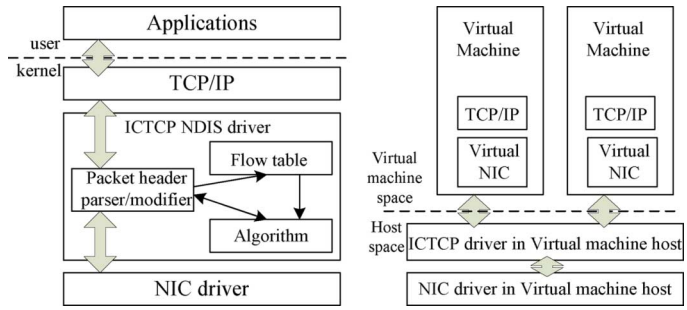


Fig. 7. Modules in ICTCP driver and software stack for virtual machine support.

Although the implementation of ICTCP in a TCP stack is natural, this paper chooses a different approach—developing ICTCP as an NDIS driver on Windows OS. The software stack is shown in Fig. 7. The NDIS ICTCP driver is implemented in the Windows kernel, between the TCP/IP and network interface card (NIC) driver, known as a Windows filter driver. Our NDIS driver intercepts TCP packets and modifies the receive window size if needed. We believe the final solution for ICTCP should be in the TCP module, while our implementation of ICTCP in an NDIS 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 virtual machines, which are widely used in data centers. We discuss this point in detail in the following sectioin. 2) ICTCP needs the incoming throughput on a very small time granularity (comparable to RTT at hundreds of microseconds) to estimate available bandwidth, and this information can be easily obtained at a driver. Note that the incoming traffic includes all types of traffic arriving on that interface, besides TCP. 3) It does not touch TCP/IP implementation in the Windows kernel. As a quick and dirty solution, it supports all OS versions instead of patching each one by one for deployment in a large data-center network with various TCP implementations.

As shown in Fig. 7, the ICTCP driver implementation contains three software modules: packet header parser/modifier, flow table, and the ICTCP algorithm. The parser/modifier implements the functions in order to both check the packet header and modify the receive window on the TCP packet header. A 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 flow information for all the active flows. For a TCP flow, its entry is removed from the table if an FIN/RST packet is observed, or no packets are parsed for 15 min. The algorithm part implements all the algorithms described in Section IV.

The operations of an ICTCP driver are as follows. 1) When the intermediate driver captures packets through either NDIS sending or receiving entry, it will redirect the packet to the

header parser module. 2) The packet header is parsed and the corresponding information is updated in the flow table. 3) The 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 the TCP header if needed.

Our ICTCP driver does not introduce extra CPU overhead for packet checksum when the receive window is modified for an outgoing ACK packet, as the checksum calculation is loaded to NIC on the hardware, which is the normal setup in a data center. There is also no packet reordering in our driver.

Although it is not the focus of this paper, we measured the CPU overhead introduced by our filter driver on a Dell server PowerEdge R200, Xeon (4CPU) at 2.8 GHz, 8 GB Memory, and compared it to the case in which our driver is not installed. The overhead is around 5%–6% for 1-Gb/s throughput, and less than 1% for 100-Mb/s throughput.

B. Support for Virtual Machines

Virtual machines are widely used in data centers. When a 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, in practice, is usually a static value. 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 will not be busy at the same time. Therefore, it creates a challenge for ICTCP in virtual machines, as the observed virtual link capacity and available bandwidth does not represent the real value.

One straightforward solution is to change the settings of virtual machine NICs and config the total capacity of all virtual NICs so that it is equal to that of the physical NIC. We focus on an alternative solution, which deploys an ICTCP driver on the virtual machine host server. The reason for such deployment is to achieve a high performance on virtual machine multiplexing. This is a special design for virtual machine cases and will not have conflicts even if ICTCP has already been integrated into TCP in the virtual machines by assuming the visualized NIC capacity is much larger than the physical one in the virtual machine host. For example, in the existing deployment, even if the physical link capacity is 1 Gb/s, the visualized NIC in the virtual machine may be configured as 10 Gb/s. The software stack of ICTCP in the virtual machine host is illustrated on the right side of Fig. 7, where all TCP connections passing the physical NIC are now jointly adjusted. Note that to make the real packet header available to a virtual machine host, IPsec technology cannot be deployed in virtual machines. We treat this as a drawback of driver-based ICTCP implementation. Fortunately, to the best of our knowledge, IPsec is seldom deployed in data-center networks.

C. Obtain Fine-Grained RTT at Receiver

ICTCP is deployed at the TCP receiver side, and it requires TCP RTT to adjust the TCP receive window. From the discussion in Section II-B, we need a live RTT as it changes during the connection, even if there is a single connection.

The RTT can be measured at the sender side by the time elapsed between when a data packet was sent and the ACK for that data packet arrived. However, ICTCP assumes it is only implemented on the receiver side so that such an RTT is not available to the TCP receiver. We define the reverse RTT as the RTT obtained on the TCP receiver side. Similar to the RTT on the sender side, we use an exponential filter to smooth the RTT samples. We consider there are two approaches to obtain the live RTT at the receiver side: 1) if the traffic between the sender and receiver is bidirectional, then the receiver can obtain the RTT; 2) by using the TCP timestamp option, the receiver can obtain the RTT by assuming the TCP sender will transmit data immediately after receiving TCP ACKs. Considering that the data traffic in the reverse direction may not be enough to keep obtaining live reverse RTT, we use the TCP timestamp option to obtain the RTT in the reverse direction. In our experiment, we observe that the reverse RTT is close to the RTT exponentially filtered at the TCP sender side.

Unfortunately, the RTT implementation in the existing TCP module uses a clock with millisecond granularity. To obtain an accurate RTT for ICTCP in a data-center network, the granularity should be in microseconds. Therefore, we modify the timestamp counter to 100-ns granularity to obtain live and accurate RTT. Note that this modification does not introduce extra overhead as such a fine-grain time is already available on the Windows kernel. We believe that a similar approach can be taken in other OSs. Our change of time granularity on the TCP timestamp follows the requirements of RFC1323 [13]. In addition, such modification also happens only on the receiver side, as a TCP sender just needs to echo the timestamp from the receiver normally and the timestamp from the sender is independent of our modification.

VII. Experimental Results

We deployed a testbed with 47 servers and one Quanta LB4G 48-port Gigabit Ethernet switch. The topology of our testbed was the same as the one shown on the right side of Fig. 1, where 47 servers each connect to the 48-port Gigabit Ethernet switch with a Gigabit Ethernet interface. Each server has two 2.2-GB Intel Xeon CPUs E5520 (four cores), 32 GB RAM, a 1-TB hard disk, and one Broadcom BCM5709C NetXtreme II Gigabit Ethernet NIC.

The OS on 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 VI and a state-of-the-art TCP New Reno with SACK implementation on a Windows server. The default timeout value of TCP on a Windows server is 300 ms. Note that all the TCP stacks were the same in our experiments, and ICTCP was implemented on a filter driver at the receiver side.

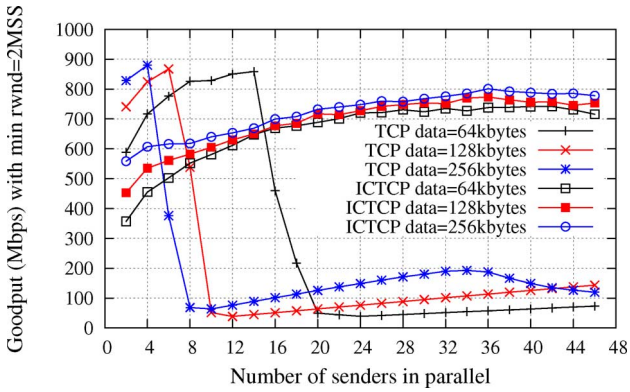


Fig. 8. Total goodput of multiple barrier-synchronized ICTCP/TCP connections versus the number of senders, where the data traffic volume per sender is a fixed amount.

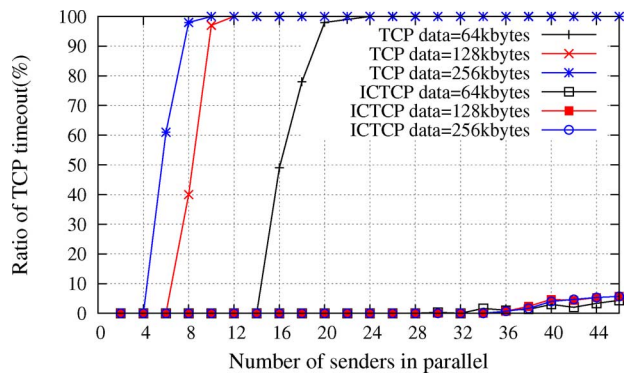


Fig. 9. Ratio of experimental rounds that suffer at least one timeout.

A. Fixed Traffic Volume per Server With the Number of Senders Increasing

The first incast scenario we considered was one in which a number of senders generate the same amount of TCP traffic to a specific receiver under the same switch. Similar to the setup in [1] and [14], we fix the traffic amount generated per sending server.

The TCP connections are barrier-synchronized per round, i.e., one round finishes only after all TCP connections in it have finished, and then the next round starts. The goodput shown is the average 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: With a larger amount of data, a smaller number of sending servers is required to trigger incast congestion.

1) ICTCP With Minimal Receive Window at 2MSS: Under the same setup, the performance of ICTCP is shown in Fig. 8. We observe that ICTCP achieves smooth and increasing goodput with the number of sending servers increasing. A larger data amount per sending server results in a slightly higher goodput. The averaged goodput of ICTCP shows that incast congestion is effectively throttled. The goodput of ICTCP, with a varying number of sending servers and traffic amount per sending servers, shows 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 879 Mb/ps while ICTCP achieves 607 Mb/s with four sending servers at 256 kB per server. There are two reasons: 1) During the connection initiation phase (slow start), ICTCP increases the receive window slowly than TCP increases the congestion window. Actually, ICTCP doubles the receive window at least every two RTTs, while TCP doubles its congestion window every RTT. Otherwise, ICTCP increases the receive window by 1 MSS when the available bandwidth is low. 2) The traffic amount per sending server is very small, and thus the time taken in the "slow-start" dominates the transmission time if incast congestion does not occur. Note that the low throughput of ICTCP during the initiation phase does not affect its throughput during the stable phase in a larger timescale, e.g., hundreds of milliseconds, which will be evaluated in Section VII-D.

To evaluate the effectiveness of ICTCP in avoiding timeouts, 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 Fig. 9. We observe that TCP suffers at least one timeout when incast congestion occurs, while the highest ratio for ICTCP experiencing timeout is 6%. Note that the results in Fig. 9 show that ICTCP is better than DCTCP[5], as DCTCP quickly downgrades to the same as TCP when the number of sending servers is over 35 for the static buffer. The reason that ICTCP effectively reduces the probability of timeouts is that ICTCP avoids congestion and increases the receive window only if there is enough available bandwidth on the receiving server. DCTCP relies on ECN to detect congestion, so a larger (dynamic) buffer is required to avoid buffer overflow during control latency, i.e., the time before control takes effect.

2) ICTCP With Minimal Receive Window at 1MSS: ICTCP has a possibility (although very small) to timeout since we use a 2MSS minimal receive window. In principle, with the number of connections becoming 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 a minimal receive window at 1MSS is shown in Fig. 10. We observe that timeout probability is 0, while the averaged throughput is lower than those with a 2MSS minimal receive window. For example, for 40 sending servers with 64 kB per server, the goodput is 741 Mb/s for 2MSS as shown in Fig. 8, while it is 564 Mb/s for 1MSS as shown in Fig. 10. Therefore, the minimal receive window is a tradeoff

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

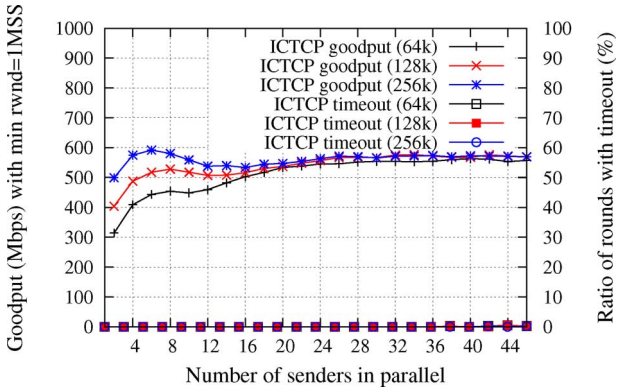


Fig. 10. ICTCP goodput and ratio of experimental rounds suffer at least one timeout with a minimal receive window of 1MSS.

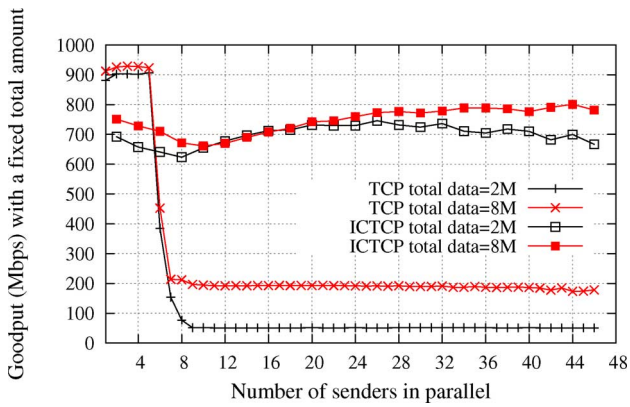


Fig. 11. Goodput of ICTCP (with a minimal receive window of 2MSS) and TCP in the case that the total data amount from all sending servers is a fixed value.

between a higher average incast goodput and a lower timeout probability.

Note that the goodput here only lasts for a very short time, $40 * 64\text{ k} * 8/564\text{ Mb/s} = 36$ ms. For a larger data size request and a longer connection duration, ICTCP actually achieves a goodput that is close to link capacity, which is shown in detail in Section VII-D.

B. Fixed Total Traffic Volume With the Number of Senders Increasing

The second scenario we consider is the one discussed in [2] and [5], 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 Figs. 11 and 12. From Fig. 11, we observe that the number of sending servers to trigger incast congestion is close for total traffic of 2 and 8 MB, respectively. ICTCP greatly improves the goodput and controls timeouts well. Note that we show the case in which ICTCP has a minimal receive window of 2MSS and skip the case of 1MSS, as the timeout ratio is again 0% for 1MSS.

C. Incast With High Throughput Background Traffic

In previous experiments, we do not explicitly generate long-term background traffic for incast experiments. Note that there is some small background traffic in our testbed from other users.

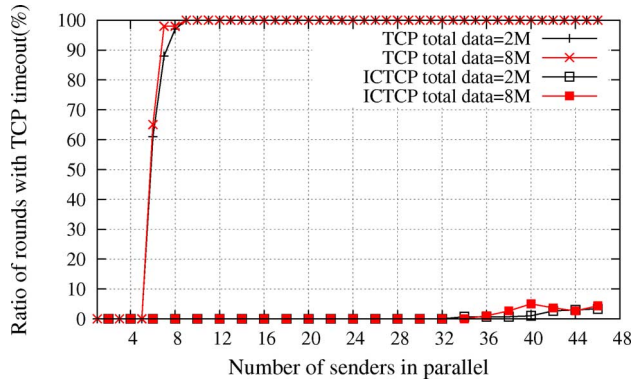


Fig. 12. Ratio of timeout for ICTCP (with a minimal receive window of 2MSS) and TCP in the case that the total data amount from all sending servers is a fixed value.

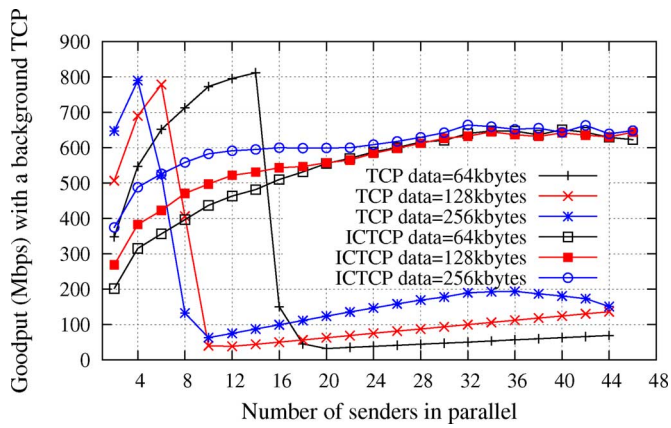


Fig. 13. Goodput of ICTCP (with a minimal receive window at 2MSS) and TCP under the case with a background long-term TCP connection.

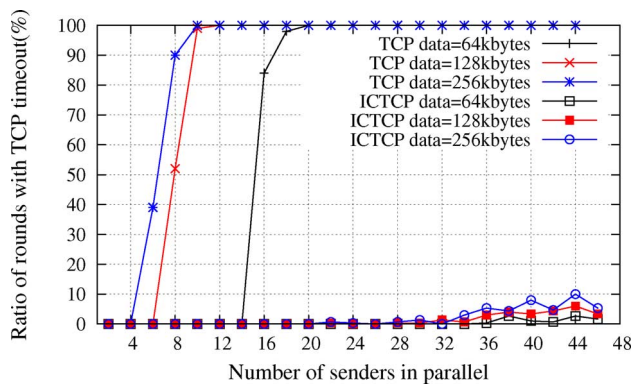


Fig. 14. Ratio of timeout for ICTCP (with a minimal receive window at 2MSS) and TCP in the case with a background long-term TCP connection.

In the third scenario, we generate a long-term TCP connection as background traffic to the same receiving server, and it occupies 900 Mb/s before incast traffic starts. Note that the background TCP lasts longer, and its receive window is also managed by ICTCP, as ICTCP has no preknowledge of which TCP connection is on incast.

The goodput and timeout ratio of TCP and ICTCP are shown in Figs. 13 and 14. Comparing Fig. 13 to Fig. 3, the throughput achieved by TCP before incast congestion is slightly lower. ICTCP also achieves slightly lower throughput when the number of sending servers is small. Comparing Fig. 14 to Fig. 9,

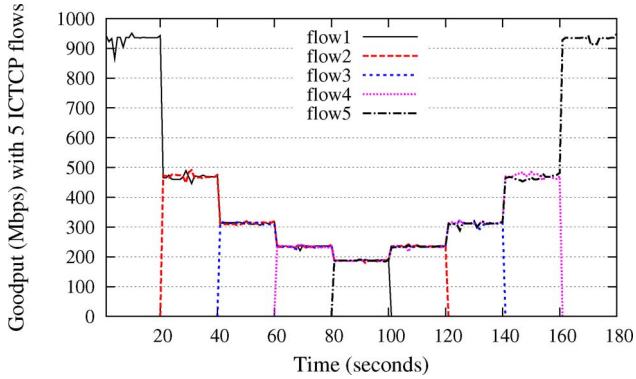


Fig. 15. Goodput of five ICTCP flows that start sequentially with 20-s intervals and 100 s duration, from five sending servers and to the same receiving server under the same switch.

the timeout ratio for 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. The throughput of the background TCP using ICTCP is not as affected as that of TCP, as the overflow probability is very small. We also obtain the experimental results for a background UDP connection at 200 Mb/s, and ICTCP also performs well. We skip the results for the background UDP as ICTCP performs similarly well.

D. Fairness and Long-Term Performance of ICTCP

To evaluate the fairness of ICTCP on multiple connections, we generate five ICTCP flows to the same receiving server under the same switch. The flows are started sequentially with 20-s intervals and 100 s duration. The achieved goodput of these five ICTCP flows is shown in Fig. 15. 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 1 Gb/s. Note that the goodput here is much larger than that shown in Fig. 8 since the traffic volume is much larger and thus a slightly higher time cost in the slow start phase is not an issue. We also run the same experiments for TCP. TCP achieves the same performance up to five servers and then degrades with more connections.

VIII. Discussions

In this section, we discuss three issues related to the further extension of ICTCP.

The first issue regards the scalability of ICTCP, in particular, how to handle incast congestion with an extremely large number of connections. Reference [5] shows that the number of concurrent connections to a receiver of 50 ms duration is less than 100 for the 90th percentile in a real data center. ICTCP can easily handle a case of 100 concurrent connections with 1MSS as the minimum receive window.

In principle, if the number of concurrent connections becomes extremely large, then we require a much smaller minimal receive window to prevent buffer overflow. However, directly using a receive window less than 1 MSS may degrade performance greatly. We propose an alternative solution: switching the receive window between several values to effectively achieve a smaller receive window averaged for multiple RTTs. For example, a 1MSS window for one RTT and a 0 window for another RTT could achieve a 0.5MSS window on average for 2RTT. Note that it still needs coordination between multiplexed flows at the receiver side to prevent concurrent connections overflow buffers.

The second issue we consider is how to extend ICTCP to handle congestion in general cases where the sender and the receiver are not under the same switch and the bottleneck link is not the last hop to the receiver. ECN can be leveraged to obtain such congestion information. However, it differs from the original ECN, which only echoes the congestion signal on the receiver side, because ICTCP can throttle the receive window considering the aggregation of multiple connections.

The third issue is whether ICTCP will work for future high-bandwidth low-latency networks. A big challenge for ICTCP is that the bandwidth may reach 100 Gb/s, while the RTT may not decrease by much. In this case, the BDP is enlarged, and the receive window on incast connections also becomes larger. While in ICTCP, a 1MSS reduction is used for window adjustment, requiring a longer time to converge if the window size is larger. To make ICTCP work for a 100-Gb/s or even higher bandwidth network, we consider the following solutions: 1) the switch buffer should be enlarged correspondingly; 2) the MSS should be enlarged so that the window size with regard to the MSS number does not enlarge greatly. This is reasonable as a 9-kB MSS is available for Gigabit Ethernet.

IX. Related Work

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

Nagle *et al.* [6] describe TCP incast in scenarios involving distributed storage cluster. TCP incast occurs when a client requests multiple pieces of data blocks from multiple storage servers in parallel. In [7], several application-level approaches were discussed. Among those approaches, they mentioned that a smaller TCP receiver buffer can be used to throttle data transfer, which is also suggested in [6]. Unfortunately, this type of static setting on the TCP receiver buffer size is at the application-level and faces the same challenge with regards to 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 [2]. Since a data center needs to support a large number of applications, a solution at the transport layer is preferred. In [1], several approaches at TCP-level have been discussed, focusing on TCP timeouts that dominate performance degradation. They show that several techniques, including alternative TCP implementations like SACK, a reduced duplicate ACK threshold, and disabling TCP slow start, cannot eliminate TCP incast congestion collapse.

Reference [2] presents a practical and effective solution to reduce the impact caused by incast congestion. Their method is to enable microsecond-granularity TCP timeouts. Faster retransmission is reasonable since the base TCP RTT in a data-center

network is only hundreds of microseconds (Fig. 5). Compared to existing widely used millisecond-granularity TCP timeouts, the interaction of fine granularity timeouts and other TCP schemes is still under investigation [14]. The major difference with our work and theirs is that our goal is to avoid packet loss, while they focus on how to mitigate the impact of packet loss, with either less frequent timeouts or faster retransmission on timeouts. This makes our work complementary to previous work, as ours is in a different solution space from theirs.

Motivated by the interaction between short flows that require short-latency and long background throughput-oriented flows, DCTCP [5] focuses on reducing the Ethernet switch buffer occupation. In DCTCP, ECN with modified thresholds is used for congestion notification, while both the TCP sender and receiver are slightly modified for a novel fine-grained congestion window adjustment. Reduced switch buffer occupation can effectively mitigate potential overflow of incast in more general cases, i.e., not only the last hop. Their experimental results show that DCTCP outperforms TCP for TCP incast, but eventually converges to an equivalent performance as the incast degree increases, e.g., over 35 senders. The difference between ICTCP and DCTCP is that ICTCP only touches the TCP receiver, and ICTCP uses the throughput increase estimation to predict whether the available bandwidth is enough for the receive window increase, thus proactively avoiding congestion.

To this end, ICTCP is in a different solution space from previous approaches addressing incast congestion. Enlarging the switch buffer size can mitigate incast congestion with a small number of senders, so it can be regarded as an intermediate switch modification only approach. Reference [2] is a sender-only approach, as it only touches the TCP retransmission timer on the sender side. DCTCP is a hybrid approach, as it changes the congestion information delivered by the ECN bit and also the actions at both the TCP sender and receiver. ICTCP is a TCP receiver side modification only solution, and thus it is complementary to previous incast approaches in different solution spaces.

To avoid congestion, TCP Vegas[11] has been proposed. In TCP Vegas, the sender adjusts the congestion window based on the difference of expected throughput and actual throughput. However, there are several issues when applying TCP Vegas directly to a receiver window based congestion control in a high-bandwidth low-latency network: 1) As we have shown in Fig. 5, 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 means that Vegas may overestimate throughput. 2) In Vegas, the absolute difference of expected and actual throughput is used. Throughput changes greatly at the granularity of RTT, as queueing latency (also at a scale of hundreds of microseconds) greatly affects the RTT sample. This makes the absolute difference defined in Vegas hard to use for window adjustment, and this is the reason we use the ratio of throughput difference over expected throughput.

The TCP receiver buffer [15], receive window, and delayed ACK [16] are used to control bandwidth sharing among multiple TCP flows on the receiver side. Previous work focused on the ratio of achieved bandwidth of those multiple independent TCP flows. Our focus is on congestion avoidance to prevent packet loss in incast, which has not been considered in previous work. Meanwhile, ICTCP adjusts the receive window for better fairness among multiple TCP flows only when available bandwidth is small. In ICTCP, we decrease the receive window by one MSS with an ACK packet to make sure that a TCP receiver will not shrink the window [12].

At the application layer, socket buffer auto-sizing (SOBAS) is proposed for improving the performance of TCP for bulk-data transfer in a grid computing scenario [17]. SOBAS is designed for a single TCP connection, and the target is avoiding repetitious switch buffer overflows introduced by TCP's congestion window maintenance. SOBAS explicitly uses the TCP throughput degradation phenomena caused by switch buffer overflow to guild later socket buffer tuning, while ICTCP uses the available bandwidth to avoid packet losses in advance.

In [18], we present the idea of ICTCP for data-center networks. In this paper, we systematically deduce the reasons for incast congestion in Section II and also provide a brief analysis to show that the switch buffer occupation under ICTCP is reasonable for an existing low-end Ethernet switch in Section V.

X. Conclusion

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

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

References

[1] 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, Article no. 12.

[2] 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. ACM SIGCOMM*, 2009, pp. 303–314.

[3] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken, "The nature of data center traffic: Measurements & analysis," in *Proc. IMC*, 2009, pp. 202–208.

[4] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *Proc. OSDI*, 2004, p. 10.

[5] M. Alizadeh, A. Greenberg, D. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data center TCP (DCTCP)," in *Proc. SIGCOMM*, 2010, pp. 63–74.

[6] D. Nagle, D. Serenyi, and A. Matthews, "The Panasas ActiveScale storage cluster: Delivering scalable high bandwidth storage," in *Proc. SC*, 2004, p. 53.

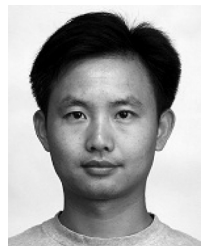
[7] E. Krevat, V. Vasudevan, A. Phanishayee, D. Andersen, G. Ganger, G. Gibson, and S. Seshan, "On application-level approaches to avoiding TCP throughput collapse in cluster-based storage systems," in *Proc. Supercomput.*, 2007, pp. 1–4.
[8] C. Guo, H. Wu, K. Tan, L. Shi, Y. Zhang, and S. Lu, "DCell: A scalable and fault tolerant network structure for data centers," in *Proc. ACM SIGCOMM*, 2008, pp. 75–86.
[9] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *Proc. ACM SIGCOMM*, 2008, pp. 63–74.
[10] 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. ACM SIGCOMM*, 2009, pp. 63–74.
[11] L. Brakmo and L. Peterson, "TCP Vegas: End to end congestion avoidance on a global internet," *IEEE J. Sel. Areas Commun.*, vol. 13, no. 8, pp. 1465–1480, Oct. 1995.
[12] R. Braden, "Requirements for internet hosts—Communication layers," RFC1122, Oct. 1989.
[13] V. Jacobson, R. Braden, and D. Borman, "TCP extensions for high performance," RFC1323, May 1992.
[14] Y. Chen, R. Griffith, J. Liu, R. Katz, and A. Joseph, "Understanding TCP incast throughput collapse in datacenter networks," in *Proc. WREN*, 2009, pp. 73–82.
[15] N. Spring, M. Chesire, M. Berryman, and V. Sahasranaman, "Receiver based management of low bandwidth access links," in *Proc. IEEE INFOCOM*, 2000, vol. 1, pp. 245–254.
[16] P. Mehra, A. Zakhor, and C. Vleeschouwer, "Receiver-driven bandwidth sharing for TCP," in *Proc. IEEE INFOCOM*, 2003, vol. 2, pp. 1145–1155.
[17] R. Prasad, M. Jain, and C. Dovrolis, "Socket buffer auto-sizing for high-performance data transfers," *J. Grid Comput.*, vol. 1, no. 4, pp. 361–376, 2003.
[18] H. Wu, Z. Feng, C. Guo, and Y. Zhang, "ICTCP: Incast congestion control for TCP in data center networks," in *Proc. CoNEXT*, 2010, Article no. 13.

Haitao Wu (M'03) received the Bachelor's degree in telecommunications engineering and Ph.D. degree in telecommunications and information systems from Beijing University of Posts and Telecommunications (BUPT), Beijing, China, in 1998 and 2003, respectively.

He joined the Wireless and Networking Group, Microsoft Research Asia (MSRA), Beijing, China, in 2003. His research interests include data-center networks, QoS, TCP/IP, and wireless networks.

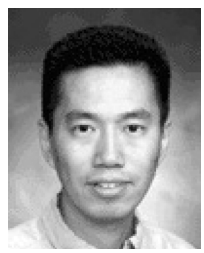
Zhenqian Feng received the Bachelor's and Master's degrees in computer science and technology from the National University of Defense Technology (NUDT), Changsha, China, in 2005 and 2008, respectively, and is currently pursuing the Ph.D. degree in computer science and technology at the same university.

His research interests include data-center networks, TCP/IP, and network file systems.



Chuanxiong Guo (M'03) received the Ph.D. degree in communications and information systems from Nanjing Institute of Communications Engineering, Nanjing, China, in 2000.

He is a Senior Researcher with the Wireless and Networking Group, Microsoft Research Asia, Beijing, China. His research interests include network systems design and analysis, data-center networking, data-centric networking, network security, networking support for operating systems, and cloud computing.



Yongguang Zhang (M'94–SM'11) received the Ph.D. degree in computer science from Purdue University, West Lafayette, IN, in 1994.

He is a Principle Researcher and Research Manager with the Wireless and Networking Group, Microsoft Research Asia, Beijing, China. Before joining Microsoft in early 2006, he was a Senior Research Scientist with HRL Labs, Malibu, CA. He was also an Adjunct Professor of computer science with the University of Texas at Austin from 2001 to 2003. He has published over 50 technical papers and one book

Dr. Zhang was an Associate Editor for the IEEE TRANSACTIONS ON MOBILE COMPUTING and a General Co-Chair for ACM MobiCom 2009.