Taming TCP Incast Throughput Collapse in Data Center Networks

Jiao Zhang, Fengyuan Ren, Li Tang, and Chuang Lin
Dept. of Computer Science and Technology, Tsinghua University, Beijing, China
Tsinghua National Laboratory for Information Science and Technology, Beijing, China
{zhangjiao08, renfy, tangli, clin}@csnet1.cs.tsinghua.edu.cn

***Abstract*—The TCP incast problem attracts a lot of attention due to its wide existence in cloud services and catastrophic performance degradation. Some effort has been made to solve it. However, the industry is still struggling with it, such as Facebook. Based on the investigation that the TCP incast problem is mainly caused by the TimeOuts (TOs) occurring at the boundary of the stripe units, this paper presents a simple and effective TCP enhanced mechanism, called GIP (Guarantee Important Packets), for the applications with the TCP incast problem. The main idea is making TCP aware of the boundaries of the stripe units, and reducing the congestion window of each flow at the start of each stripe unit as well as redundantly transmitting the last packet of each stripe unit. GIP modifies TCP a little at the end hosts, thus it can be easily implemented. Also, it poses no impact on the other TCP-based applications. The results of both experiments on our testbed and simulations on the ns-2 platform demonstrate that TCP with GIP can avoid almost all of the TOs and achieve high goodput for applications with the incast communication pattern.**

Keywords*—*Data Center Networks, TCP incast, Goodput, Initial Window, Redundant Transmission

I. Introduction

The TCP incast problem occurs when multiple senders synchronously transmit stripe units to a single receiver in data center networks with high bandwidth and low Round-Trip Time (RTT) [1]–[3]. Any sender can not transmit the next stripe unit until all the senders finish transmitting the current ones. The synchronized transmissions with this restriction is called *barrier traffic pattern* [2], [17]. When the number of the concurrent senders becomes large, the shallow switch buffer [13], [14] is likely overwhelmed. If the retransmission mechanism of TCP fails to recover the large amount of spilled packets, TO periods will occur. The Minimum Retransmission TimeOut (`RTOmin`) of TCP generally equals 200 milliseconds in default, which is orders of magnitude of the microsecond-granularity RTT in data center networks, so a large number of TO periods will dramatically decrease the goodput of TCP.

The TCP incast problem attracts a lot of attention because of the catastrophic goodput drop and the wide existence of the incast communication pattern in big data systems. First, companies offering online services generally store immense amount of data using the distributed storage techniques, such as BigTable [4], HBase [5], Hypertable [6]. When a client retrieves data, parallel access to some of the distributed storage nodes is required. The TCP incast throughput collapse is firstly found in the distributed storage system PanFS [7]. In Facebook, dramatic goodput reduction also happens when the clients retrieve data from the memcached system. Second, the Data-Intensive Scalable Computing Systems (DISC) [8], such as MapReduce [9], Dryad [10], Spark [11], achieve high-speed computing through the cooperation of many distributed servers. Thus, many-to-one transmissions are needed to transfer data from mappers to reducers during the *shuffle* phase. A large number of packets are easily dropped at the switches that connect to the reducers and thus the TCP incast problem occurs [12]. Similarly, most of today's large-scale, user facing web applications follow the partition/aggregation design pattern [13], [14]. Large-scale tasks are divided into pieces and assigned to different workers. All the responses from the workers are aggregated to generate the final result. This pattern improves the response speed. However, the aggregation period naturally leads to the incast problem at the aggregators.

Some recent mechanisms can be employed to solve the TCP incast problem. For example, several recently proposed latency-aware transport protocols can avoid the TCP incast problem by employing some switch mechanisms to reduce packet losses [13]–[15]. However, possibly because it is a little hard to implement switch-based mechanisms in practice, the industry is still struggling with the TCP incast problem. For example, Facebook engineers proposed limiting the number of parallel requests to alleviate the problem in their recent published work [16]. Considering that TCP is widely used in production data centers [13], this paper aims to solve the TCP incast problem by modifying TCP a little at the end hosts.

The typical end-host based solutions to the TCP incast problem are decreasing `RTOmin` [1] and ICTCP [17]. `RTOmin` of 200 milliseconds is proper for the traditional internet whose RTT is in the granularity of milliseconds, but too large for data center networks. Reducing it can diminish the bandwidth wastage during TO periods. However, if the bandwidth capacity increases [18], [19], even if microsecond-granularity TOs will bring large penalty to goodput. Besides, too small `RTOmin` may cause spurious retransmissions, especially in the external (at least one end point is outside the data center) connections with larger RTT. ICTCP [17] prevents TOs by dynamically adjusting the Advertised Window (`awnd`) at the receiver side. However, it only focuses on the incast scenarios where the last hop is the bottleneck.

Through observing lots of trace data in incast scenarios, we found that two types of TOs should be avoided to improve the goodput. First, the TOs caused by full window losses. When the number of senders becomes large, the bandwidth occupied by each sender is small. Possibly the packets in a full window of an unlucky flow will be dropped because of severe traffic bursts induced by the synchronized transmissions. This kind of TOs is denoted as Full window Loss TimeOut (FLoss-TO). Second, the TOs caused by Lack of ACKs (LAck-TO). Since the receiver will not request the senders to transmit the next stripe units until all of the senders finish their current ones, once some packets are dropped at the tail of a stripe unit, the Fast Retransmission or Fast Recovery (FR/FR) is hardly triggered due to inadequate ACKs. Thus the lost packets will not be recovered until the retransmission timer fires.

In this paper, two mechanisms are proposed to eliminate these two kinds of TOs. One is reducing the initial congestion window of each sender at the head of each stripe unit to avoid FLoss-TOs. Another is redundantly transmitting the last packet of a stripe unit to avert the LAck-TOs. Since the enhanced mechanism avoids the two kinds of TOs mainly through Guaranteeing Important Packets (GIP) not dropping, we referred to this combined mechanism as TCP with GIP in the rest of this paper.

We implement the GIP mechanism in CentOS-5.5 with Linux kernel version 2.6.18. TCP with GIP follows the basic congestion control mechanism of TCP, and employs the `flags` in the interface between the application layer and the TCP layer to indicate whether the running application has the incast communication pattern or not. For incast applications, TCP with GIP works, that is, the last packet of a stripe unit will be redundantly transmitted at most three times and each sender decreases its initial congestion window at the head of each stripe unit. While for other applications, the standard TCP works. Thus, TCP with GIP keeps backward compatibility.

An experimental testbed is deployed with 24 Dell servers and Gigabit Ethernet switches. The experimental results validate that the TCP with GIP can avoid the TOs and thus solve the incast throughput collapse. Also, it is able to adapt to more network scenarios than ICTCP does since the latter can only work when the link connecting to the receiver is the bottleneck [17]. Furthermore, to evaluate the performance of TCP with GIP in data centers with higher bandwidth and more senders, series of simulations are conducted on the ns-2 platform. The results demonstrate that TCP with GIP has good scalability compared with the algorithm of reducing `RTOmin`.

From a practical point of view, the proposed TCP with GIP has three advantages. First, it poses no impact on the other TCP connections, including both the internal and the external ones. Second, it does not rely on the OS version since it does not need any special functions provided by the kernel, such as high resolution timer. At last, it is very simple. Only dozens of lines of codes are added to the TCP protocol.

The paper is organized as follows. The prior solutions to the TCP incast problem and their features are summarized in Section II. In Section III, the causes of the TCP incast problem are analyzed. Section IV shows the GIP mechanism in detail. Section V depicts the implementation of TCP with GIP. In Section VI, the performance of TCP with GIP is evaluated on our deployed experimental testbed, and compared with NewReno. In Section VII, we conduct simulations on the ns-2 platform with higher bandwidth and larger number of senders, and compare the performance of TCP with GIP with TCP NewReno and `RTOmin=2` milliseconds. Section VIII discusses the limitation and rationality of TCP with GIP. Finally the paper is concluded in Section IX.

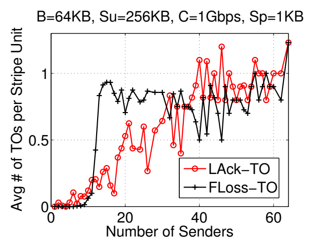
II. Related work

Since the TCP incast problem was proposed by Nagle *et al.* in [7], some work has been done to address it.

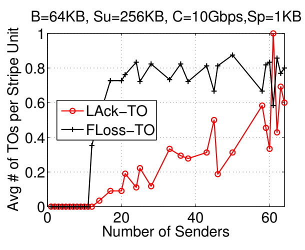
Modify TCP. Since the incast goodput collapse is caused by a large number of TO periods, some attempts are made to avoid the timeouts, such as trying different TCP versions, enabling Limited Transmit [20], reducing duplicate ACK threshold, disabling TCP slow start [2], [3]. However, V. Vasudevan *et al.* claimed that some TO periods in the incast scenarios are hard to be eliminated without extra mechanisms, so they suggested reducing `RTOmin` to alleviate the goodput decline [1]. But this method poses implementation challenges and may cause safety problems such as spurious retransmission [21]. The above methods attempt to modify TCP at the sender side. Yet the number of senders is possibly large in the incast applications. Each sender is difficult to get enough information to adjust its congestion window properly. So H. Wu *et al.* proposed ICTCP [17] which controls flow rate by adaptively adjusting the `awnd` at the receiver side. The receiver estimates the available bandwidth and RTT to compute the reasonable `awnd`. However, exact estimation of the real-time available bandwidth and the time-varying RTT is difficult. And foremost, ICTCP fails to work well if the bottleneck is not the link that connects to the receiver.

Design New Transmission Protocol. M. Alizadeh *et al.* observed the traffic characteristics in data center networks and stated that to satisfy the requirements of long and short flows and solve the TCP incast problem, the queue length must be persistently low. They proposed DCTCP [13] which utilizes the Explicit Congestion Notification (ECN) and revises the TCP congestion control mechanism at the source to maintain a small switch queue length and thus avoid timeouts. However, the experimental results shown in [13] indicate that DCTCP can not deal with the incast problem when the number of flows is relatively large. D^3 [14] and PDQ [15] provide delay-aware transmission control by explicit rate control at switches and emulating preemptive scheduling mechanisms, respectively. However, it is too costly for all companies to substitute the widely used TCP with a new transmission protocol, especially with switch modification, only because of the TCP Incast problem.

Employ Congestion Control Protocol at the Link Layer. A. Phanishayee *et al.* tried to solve the TCP throughput collapse by enabling Ethernet Flow Control (EFC) [2]. The results look good but most switch vendors disable EFC due to head-of-line blocking. P. Devkota *et al.* suggests preventing packets dropping by modifying Quantized Congestion Notification (QCN) [22], which is a link layer congestion control mechanism designed for data centers. The authors found that QCN proposed by IEEE 802.1 qau group could not solve TCP incast collapse by observing simulation results. Thus, they modified QCN by increasing the sampling frequency at the Congestion Point (CP) and making the link rate increase adaptively to the number of flows at the Reaction Point (RP). However, the modified QCN introduces overhead for all the other applications running in data center networks. Recently, Y. Zhang *et al.* proposed another modified QCN mechanism, fair QCN (FQCN), to mitigate the TCP Incast collapse in data centers [23]. They found that the low TCP throughput in an incast scenario with QCN is caused by the unfairness among the flows. Therefore, they modified the congestion feedback in QCN to improve the fairness of different flows along the same bottleneck. However, FQCN requires switches to monitor the packet arrival rate of each flow, which incurs high overhead in data centers with many concurrent flows. What's more, QCN will cause collateral damage to the flows which do not share



(a) 1 Gbps



(b) 10 Gbps

Fig. 1. Avg. number of LAck-TOs and FLoss-TOs with different link bandwidth.

the bottleneck link [13].

III. Causes of TCP Incast Problem

Examining lots of simulation and experimental data, we found that two main categories of TOs lead to the TCP incast problem. One is LAck-TO, which is caused by insufficient packets for data-driven recovery at the tail of stripe units. For example, if a flow losses the last packet of its current stripe unit, then the only recovery method is waiting for timeout since the incast application will not deliver new data to the TCP layer until all the current stripe units are finished. The other one is FLoss-TO, which is induced by the full window loss when the traffic burst is too heavy at the start of a stripe unit. The congestion window of each flow at the end of current stripe units could be quite different and the summation of them may be much larger than the bottleneck capacity. Possibly some unlucky flows with small congestion windows will loss the packets of the full windows.

Figures 1(a) and 1(b) draw the average number of different kinds of TOs suffered by the most unfortunate flow per stripe unit when the bottleneck link capacity is 1 Gbps and 10 Gbps, respectively. The bottleneck buffer $B = 64$ KB, the packet size is 1 KB, and the stripe unit size (S_u) is 256 KB. Figure 1(a) exhibits that when C=1 Gbps, most of the TOs are caused by inadequate ACKs as the number of senders is smaller than 12, while the TOs caused by the full window losses take majority as the number of senders ranges from 12 to 30. After 30, the two kinds of TOs are almost the same. Figure 1(b) shows that when C becomes larger, both LAck-TOs and FLoss-TOs rarely happen as the number of senders is small, and FLoss-TOs plays a leading role when the number of senders is large.

The theoretical model also indicates that the incast throughput collapse is indeed mainly caused by LAck-TOs and FLoss-TOs [24]. Therefore, we need to design a mechanism that can prevent the two kinds of TOs.

IV. TCP with GIP

We assume that switches employ the simple drop tail queue management scheme. The `awnd` of the receiver is large enough so as not to cap the TCP sending window. Next the basic idea of TCP with GIP is presented, following by the mechanisms of overcoming LAck-TOs and the FLoss-TOs.

A. Basic Idea

Note that TCP works well for traditional applications which will continuously deliver data to the transport layer as long as the applications have data to be sent. However, the applications with the incast communication pattern deliver traffic in stripe units. TCP layer will not receive the next stripe units until all the senders finish their current ones. This feature disrupts the normal operation of TCP. For example, the Fast Retransmit/Fast Recovery (FR/FR) mechanism of TCP is designed under the premise that after one packet being dropped, TCP has packets to send to generate three duplicate ACKs which will trigger the event of recovering the lost packet. However, in the applications with the incast communication pattern, if at least one of the last three packets of a stripe unit is dropped, then TCP does not have enough packets to send to generate sufficient duplicate ACKs. Therefore, the FR/FR mechanism of TCP does not function at the end of the stripe units and LAck-TOs happen. The FLoss-TOs are also caused by the special incast communication pattern. In [25], David D. Clark *et al.* suggested that the applications should break the data into Application Data Units (ADUs), and the lower layers preserve these frame boundaries as they process data. Enlightened by the suggestion, in our proposed TCP with GIP protocol, we let *TCP be aware of the boundaries of the stripe units and use the boundary information to deal with the special TOs.*

First, at the tail of a stripe unit, since the TOs are caused by insufficient ACKs, we let TCP transmit redundant packets to generate duplicate ACKs to prevent this kind of TOs. Then the question is which packets should be transmitted? Should some of the packet at the end of the stripe units be retransmitted? Or should some extra packets be added? Besides, how many packets should be transmitted? In Section IV-B, we will describe our solution in detail.

Second, at the head of a stripe unit, all the senders start from their saved congestion window values at the end of the last stripe unit and inject packets to the network at the same time. However, the summation of these initial congestion windows is likely larger than the network capacity. This is because some senders finish their last stripe units earlier than the others, then the remainder will take more bandwidth than $\frac{1}{N}$ of the bottleneck link capacity (Assume there are a total of N senders.). Thus, the injected packets at the start of stripe units will easily cause many packets being dropped. As the number of senders increases, the congestion window of each connection is small, thus full window losses possibly happen. To solve this problem, the initial congestion window value should be adjusted to match the bottleneck link bandwidth. In Section IV-C, we will discuss how large the initial congestion window should be.

B. Overcome LAck-TOs

1) Which lost packets cause LAck-TOs: LAck-TOs are caused by packet losses at the end of stripe units. Let the last three packets of a stripe unit be denoted as 1, 2, 3, respectively. Fig. 2 summarizes different combinations of the dropped packets that cause LAck-TOs, in a simulation scenario with 64 KB buffer size, 1 Gbps bottleneck bandwidth, and 256 KB stripe unit size. We are aware that when the number of senders is determined, each of the last three packets has the comparative loss probability, and two of them may be lost together, but the situation that all three of them are lost rarely happens.

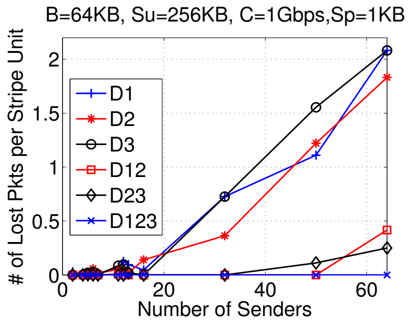


Fig. 2. Dropped packets which result in LAck-TOs.

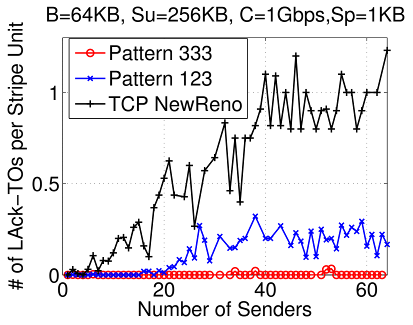


Fig. 3. Number of LAck-TOs per stripe unit.

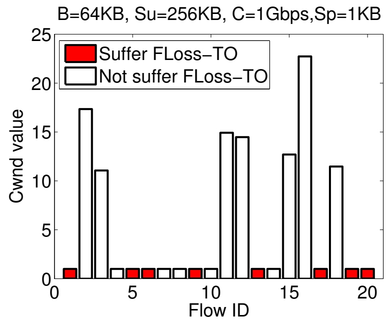


Fig. 4. `cwnd` value at the start of the 7-th stripe unit.

2) Basic idea to avoid LAck-TOs: To avoid the LAck-TOs, it is necessary to ensure all of the last three packets reach the receiver before the corresponding retransmission timer fires. Hence, once one of the last three packets is dropped, the sender should enter FR/FR period instead of waiting for the expiration of the corresponding retransmission timer. However, TCP can not enter FR/FR period without enough ACKs.

Therefore, one straightforward way is to insert several extra packets at the tail of a stripe unit to generate more ACKs. Once the receiver receives all the current stripe units, it will request the senders to transmit the next ones no matter whether the extra packets have reached the receiver or not. However, in this method, the extra packets need to be specially dealt with. For example, once they are lost, they are not required to be retransmitted to guarantee the reliability. Another method is redundantly transmitting some of the last three packets no matter whether they are lost before their retransmission timer fires. This method can not only generate more ACKs but also improve the probability that the last three packets successfully reach the receiver before LAck-TOs occur. Besides, this method can be more easily deployed since it does not need to manually generate new packets at the TCP layer.

3) Which packets should be redundantly sent: Intuitively, since LAck-TOs are caused by the loss of the last three packets, which has almost the same loss probability (see Figure 2), the last three packets should be redundantly transmitted in turn, named as *Pattern 123*, to ensure that they successfully reach the receiver without suffering LAck-TOs. Note that in this method, the Congestion Window `(cwnd)` should be decreased by 3 before the redundant retransmission. Otherwise, all the three redundantly retransmitted packets will be sent at the same time since all the packets with sequence number within `(min{cwnd, awnd}+highest_ack)` can be transmitted. Fig. 3 shows the goodput under *Pattern 123*. Unfortunately, the results of this repetitive transmission pattern are unsatisfactory. Only part of LACK-TOs can be avoided.

Investigating the trace data, we found that two main reasons lead to the unsatisfactory result under *Pattern 123*.

First, under *Pattern 123*, if only the last packet is dropped, then the receiver will see duplicate packets 1 and 2 and ignore them instead of sending back ACK. Thus, the senders can not receive sufficient ACKs to trigger fast retransmission.

Second, under *Pattern 123*, some of the redundantly transmitted packets are easily dropped. According to the window-based congestion control scheme employed by TCP, upon receiving one ACK, `cwnd` increases $\frac{1}{\texttt{cwnd}}$ in Congestion Avoidance (CA) periods. The sender then transmits the packets whose sequence number is within `(min{cwnd, awnd}+highest_ack)`. Note that `cwnd` will not increase until it has passed the next integer. If the new ACK increases the value of `cwnd` by 1, then two packets will be sent in succession. However, one ACK implies that only one packet departs from the queue, therefore the second transmission is excessive and likely to be dropped. Here is a simple example. We do not consider the `awnd` limitation at the receiver side. If the slow start threshold is 2 packets and the current `cwnd` value is 2, then after receiving a new ACK, the `cwnd` increases to be $\texttt{cwnd}+\frac{1}{\texttt{cwnd}}=2+\frac{1}{2}=2.5$ and the `highest_ack` increases by 1. Then one more packet can be sent based on the sequence number upper bound `(min{cwnd, awnd}+highest_ack)`. After receiving another ACK, `cwnd` becomes to be $2.5+\frac{1}{2.5}=2.9$. Similarly, only one packet is injected to the network. However, if one more new ACK arrives, the `cwnd` value will increase to be $2.9+\frac{1}{2.9}=3.2448$, that is, the `cwnd` passes the next integer 3. Therefore, two more packets will be transmitted. Since one new ACK means the departure of one packet from the queue, the second transmitted data packet will possibly be lost. This phenomenon is explained in detail in [26].

When the `cwnd` becomes smaller as the number of senders increases, the probability that `cwnd` just passes the next integer after receiving an ACK will get larger. Therefore, under *Pattern 123*, possibly two packets will be sent in succession if the `cwnd` has just passed the next integer, which violates the pipeline model of TCP since only one ACK is received. Hence, the second packet may be dropped. Observing the simulation data, many redundantly transmitted packets are indeed dropped due to the above reason.

To avoid the drawbacks of *Pattern 123*, we can make use of the *barrier traffic pattern* of the incast applications and only redundantly transmit the last packet. As stated previously, the incast applications will not deliver the next stripe unit to the TCP layer until the current one is finished. Hence, no matter how large the sending window is, the sender transmits at most *one* packet upon receiving one ACK. Therefore, the pipeline keeps dynamically equilibrium and thus prevents packets from dropping. The probability of the successful redundant transmission can also be improved.

Then how many redundant transmissions are proper? Consider the worst situation. If the last two packets are dropped, which likely occurs (Fig. 2), then at least 3 repetitive transmissions are needed to generate enough duplicate ACKs to drive TCP to enter FR/FR procedure. Since more repetitive transmissions waste more bandwidth, the number of redundant transmissions is set to 3 in the GIP mechanism. Figure 3 shows the number of LAck-TOs per stripe unit under TCP NewReno and TCP with different retransmission patterns. We can see that few LAck-TOs happen when the last packet is repetitively transmitted 3 times, i.e., following *Pattern 333*.

C. Overcome FLoss-TOs

1) Why the packets in a full window are lost: As stated in Section IV-A, as the number of senders becomes relatively large, the asynchronism of their `cwnd` evolution causes that some of them finish their stripe units earlier. Then the others occupy the available bottleneck bandwidth to finish their remaining data. Thus, at the end of the stripe unit, the summation of the `cwnd` of *all* the flows will highly exceeds the bottleneck buffer. After they inject the packets in their whole windows synchronously at the start of the next stripe unit, lots of packets will be dropped. The unlucky flow, which loses the packets in its full window, will suffer a FLoss-TO.

The simulation data validates our analysis. For example, Fig. 4 illustrates the `cwnd` values of different flows at the beginning of the 7-th stripe unit in an incast scenario where buffer size $B = 64$ KB, stripe unit size $S_b = 256$ KB, packet size $S_p = 1$ KB and link capacity $C = 1$ Gbps. The solid bars are the `cwnd` values of the flows that suffer FLoss-TOs at the start of the 7-th stripe unit, and the hollow bars present the `cwnd` values of the flows without experiencing FLoss-TOs. Obviously, all the flows suffering FLoss-TOs have small `cwnd` (i.e., `cwnd=1`), while many of the other flows have relatively large `cwnd` values. What's more, the summation of all the flows is 117 pkts, which is far more than the network capacity, $CD + B = 72.5$ pkts. Therefore, at the beginning of the 7-th stripe unit, all the flows simultaneously inject the packets according to their sending windows, then the buffer can not accommodate all of them. The flows with small `cwnd` will likely lose the packets in a full window.

2) Avoiding full window loss: Since the full window loss is caused by the large summation and big variance of the initial `cwnd` size of the senders, one straightforward method is to reduce the initial `cwnd` of each flow at the start of each stripe unit. The value should be small enough so that the bottleneck buffer can accommodate most of the packets in the first RTT period. Besides, the `cwnd` of each flow should be identical so that each flow can fairly share the bandwidth. Therefore, in GIP, all the flows start from slow start phase at the beginning of each stripe unit to avoid FLoss-TOs.

V. IMPLEMENTATION

We implement the GIP mechanism in CentOS-5.5 whose kernel version is 2.6.18. First, we properly set some parameters to ensure that the maximum rate of TCP can reach about 1 Gbps. Due to the relatively small default window size of TCP, generally the rate of a TCP connection can not reach up to 1 Gbps even if the network capacity is large enough. We modify the default values of both `net.ipv4.tcp_wmem` and `net.ipv4.tcp_rmem` to be 128 KB, and their maximums are set to be 256 KB. `tcp_wmem` denotes the send buffer memory space allocated to each TCP connection, and `tcp_rmem` stands for the receive buffer size of each TCP connection. Then we use *Iperf* to test the maximum rate of one TCP session between two PCs connected by a Gigabit switch. The result approximates to 950Mbps.

Subsequently, we add the proposed GIP function into TCP NewReno. The implementation mainly includes three parts: differentiating the boundaries of the stripe units, redundantly transmitting the last packet of each stripe unit for at most 3 times, and forcing each flow to start from the slow start period at the start of each stripe unit. They are implemented in three functions respectively: the socket interface `send()`, functions `tcp_sendmsg()` and `tcp_rcv_established()`. Fig. 5 shows the changes. Note that, only the TCP sender side is modified.

A. Notifying the Boundaries of Stripe Units

All the application programming interfaces that are used to transfer data from the application layer to the TCP layer have a parameter `flags`, i.e., `ssize_t send (int s, const void *buf, size_t len, signed int` *`flags`*`)`. With the evolution of TCP protocols, some bits of the flags are gradually defined to indicate some special requirements. For example, `flags=0x40 (MSG_DONTWAIT)` enables non-blocking operation. Until now, the maximum defined `flags` is $0x8000$ (`MSG_MORE` since Linux kernel 2.4.4) which notifies TCP that the application has more data to send. Since the type of `flags` is `signed int`, 31 bits can be used. `MSG_MORE` uses the 15-th bit. In the GIP mechanism, the application layer sets the 16-th bit of `flags` (i.e., **`flags`** $= 0x10000$) when calling `send()` to notify the TCP layer that the data chunk in `buf` is the tail of the current stripe unit. Since the applications with the incast communication pattern transmit data in units of strip units, clearly they know the size of a stripe unit. Thus, it is easy for the applications to notify the boundaries of the stripe units.

B. Redundant Transmissions at the Tail of Stripe Units

If the 16-th bit of the `flags` is 1, TCP layer knows that the data chunk delivered by the application layer is the tail of the current stripe unit. Then after transmitting the data chunk, TCP will retransmit the last packet once if it receives a new ACK which is not for the last packet. This process repeats at most 3 times. It is implemented in the function of `tcp_rcv_established()` as shown in Fig. 5. When the above conditions of retransmitting the last packet satisfy, TCP calls `tcp_retransmit_skb()` to redundantly transmit the last packet. A variable is added to control the number of the redundant transmissions. Note that the sequence number of the transmitted packet should be larger than the maximum number of the acknowledged packets. Thus if the sender has received the ACK of the last packet, it will not transmit it any more.

C. Reducing `cwnd` at the Head of Stripe Units

In default, the delayed ACK mechanism is enabled to reduce the number of ACKs [28]. If each flow sets `cwnd` to 1 at the start of each stripe unit, the receiver will not send an ACK until the delayed ACK timer expires. Since the default value of the delayed ACK timer is 40ms, which is quite large compared with the microsecond-granularity RTTs in data center networks, we set `cwnd` of each sender to

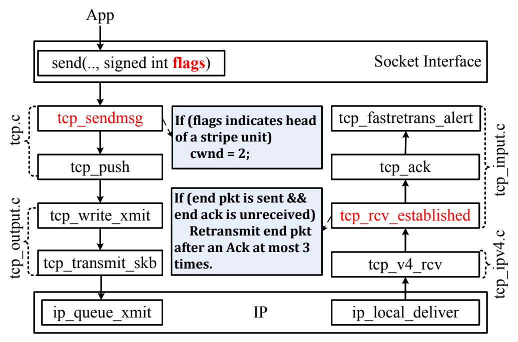


Fig. 5. TCP sender is modified in functions `tcp_sendmsg()` and `tcp_rcv_established()`. The receiver side is unchanged.

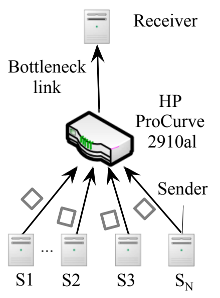


Fig. 6. The topology of TCP incast scenario.

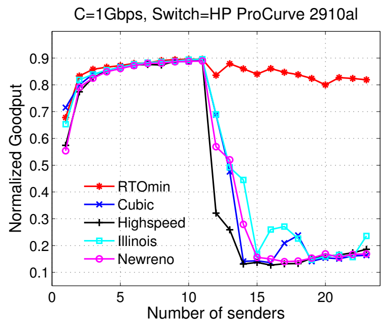


Fig. 7. Normalized goodput with different TCP versions.

2 at the start of every stripe unit to avoid the bandwidth wastage caused by the delayed ACKs. This part is implemented in the function of `tcp_sendmsg()` as shown in Fig. 5. The head of a stripe unit can be naturally known since the 16-th bit of the `flags` is used to indicate the tail of the last stripe unit. Besides, since each call of `tcp_retransmit_skb()` increases `tp->retrans_out` by 1, the value of `tcp_packets_in_flight()` will be greater than `cwnd` after transmitting several stripe units, which blocks TCP from sending packets. Thus, we reset `tp->retrans_out` to 0 at the start of each stripe unit.

In sum, the implementation of GIP is quite easy, only about 30 lines of codes are added. Besides, the enhanced mechanism GIP is backward-compatible with TCP since it has no impact on the other applications, and follows most of the TCP congestion control mechanisms. The applications can choose whether to enable the TCP GIP function by setting the 16-th bit of `flags` or not.

VI. Experimental Results

A. Experimental Setup

As shown in Fig. 6, our basic testbed is comprised of 24 servers and one HP ProCurve 2910al-48G-PoE+ Switch J9148A. Each PC is a DELL OptiPlex 360 desktop with Intel 2.93GHz dual-core CPU, 6GB DRAM, 300GB hard disk, and one Intel corporation 82567LM-3 Gigabit NIC (Network Interface Card). The operating system is CentOS-5.5.The default `RTOmin` is 200 ms.

We try several TCP variants, including Cubic, Illinois, Newreno and so on, and implement Newreno with `RTOmin=2` ms by adding the patch in [29]. The Illinois and Newreno with `RTOmin=2` ms are implemented in Linux kernel 2.6.28. The others are in kernel 2.6.18. Fig. 7 depicts their goodput results under the workload with incast communication pattern. We can see that the performance of all the TCP variants with the default `RTOmin` value is similar. Thus, we only compare our algorithm with TCP Newreno in the next subsections. Besides, we can see that the mechanism that reduces `RTOmin` to 2 ms does not suffer the TCP incast problem. Thus, we ommit the results of it in the following experiment results. However, in large scale networks with the incast communication pattern, `RTOmin=2` ms suffers many spurious timeout. Therefore, the results of it will be shown in our large scale simulations.

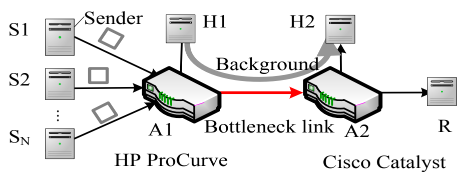


Fig. 14. The incast scenario where the intermediate link is the bottleneck.

B. Without Background Traffic

Fig. 8 shows the normalized goodput of TCP with GIP and TCP NewReno with different stripe unit sizes. The goodput of TCP NewReno collapses when the number of senders is larger than about 12, and the smaller is the stripe unit, the lower stable goodput TCP NewReno achieves. While TCP with GIP performs well as the number of senders increases. The link utilization is about 90%.

Specifically, when the number of senders is quite small, the performance of TCP with GIP is slightly worse than TCP NewReno, which mainly attributes to the mechanism of avoiding FLoss-TOs. To alleviate the traffic bursts and unfairness of different flows at the start of each stripe unit, each flow is compelled to begin with a slow start phase. Therefore, when the number of senders is quite small, the bottleneck is still under-utilized when a stripe unit is finished. However, in typical incast applications, such as MapReduce, filesystem reads, generally the number of senders is relatively large. Hence, TCP with GIP performs well in most situations.

Fig. 9 presents the average number of the TO periods suffered by the most unfortunate flow per stripe unit with different stripe unit sizes. TCP with GIP suffers few TO periods, which indicates that the GIP mechanism removes most of TOs and thus improves the goodput. Although the experiment with 256 KB stripe unit size suffers slightly more TO periods than that with 64 KB and 128 KB stripe unit, the ratio of the time consumed by the TO periods to that of transmitting a stripe unit is almost the same. Therefore, the goodput with different stripe unit size are almost the same as shown in Fig. 8.

C. With Background Traffic

1) Last hop is the bottleneck: We do not consider background traffic in the above experiments. To evaluate the

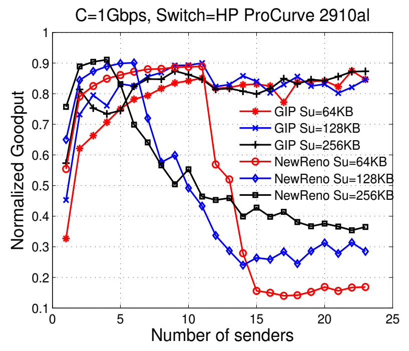


Fig. 8. Normalized goodput with different stripe unit sizes.

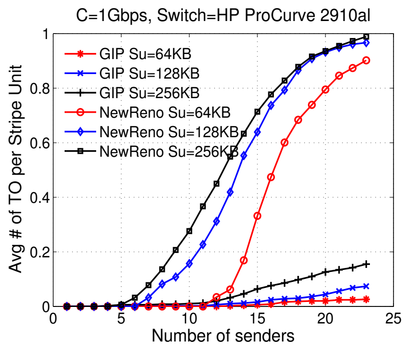


Fig. 9. Avg number of TO periods suffered by the unluckiest flow in one stripe unit.

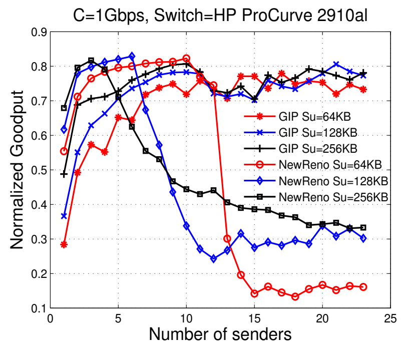


Fig. 10. Normalized goodput with different stripe unit sizes with UDP background traffic.

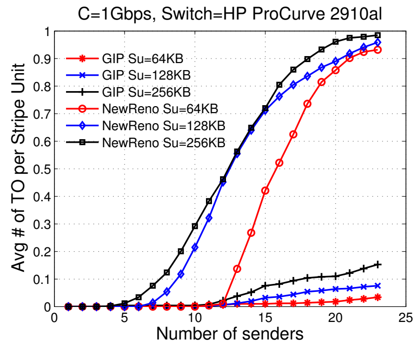


Fig. 11. Average number of TO periods suffered by the unluckiest flow with UDP background traffic.

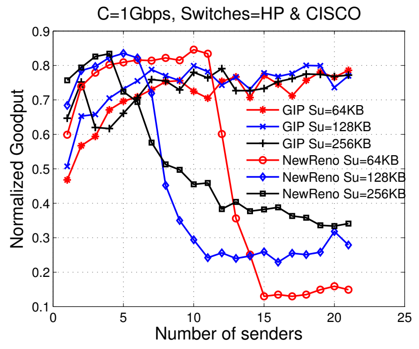


Fig. 12. Normalized goodput with UDP background traffic in multiple hops.

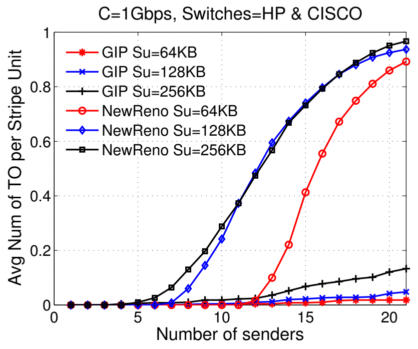


Fig. 13. Average number of TO periods suffered by the unluckiest flow with UDP background traffic in multiple hops.

performance of the GIP mechanism in incast scenarios with background traffic, a persistent UDP background traffic is injected into the bottleneck link at the rate of 100 Mbps. Figure 10 displays the normalized goodput versus the number of the senders with different stripe unit size. We can see that TCP with GIP still outperforms TCP NewReno and approaches to 0.8 Gbps goodput. Compared with Figure 8, the general tendency of TCP with GIP and TCP NewReno is similar. But the goodput is about 0.1 Gbps smaller than that in Fig. 8, since 0.1 Gbps bandwidth is consumed by the UDP background traffic.

The average number of the TO periods occurred in each stripe unit is plotted in Fig. 11. TOs happen more frequently in TCP NewReno as the number of senders increases, while GIP undergos few TO periods. The number of TOs in TCP with GIP is slightly larger than that in Fig. 9 since the background traffic contends for the bottleneck bandwidth, which results in more dropped packets.

2) Intermediate link is the bottleneck: ICTCP [17] needs to know the available bandwidth of the bottleneck to fairly allocate it among the senders by dynamically adjusting the value of `awnd`. It estimates the available bandwidth at the receiver side through observing the total incoming traffic to the receiver NIC. When the link connecting to the receiver is the bottleneck, it works well. However, when the bottleneck is one intermediate link, ICTCP fails to work normally. To verify whether TCP with GIP can adapt to this network configuration, we construct the network topology as shown in Figure 14. A Cisco Catalyst 2960G Ethernet Gigabit Switch is inserted between the HP ProCurve switch and the receiver. Server $H1$ sends UDP traffic to server $H2$ at the rate of 100 Mbps, then the link (A_1, A_2) becomes the bottleneck.

The normalized goodput is shown in Figure 12. TCP with GIP achieves about the same goodput as that in Fig. 10, which implies that the normal running of TCP with GIP is not restricted by the location of the bottleneck link, but it is the inherent limitation of ICTCP. Figure 13 reveals the average number of the TO periods suffered by the unluckiest flow per stripe unit. The number of TOs suffered by TCP with GIP and TCP NewReno is close to that in Fig. 11.

VII. Simulation

In practice, the bandwidth may be 10Gbps or even higher and one job may involve hundreds of servers [30]. To evaluate the performance of TCP with GIP with higher bottleneck capacity or larger number of senders, we implement our algorithm on the ns-2 platform and compare its performance with both TCP NewReno and `RTOmin=2` ms. The topology used in our simulations is the same as that in Fig. 6.

A. Without Background Traffic

Fig. 15 presents the goodput of TCP with GIP, `RTOmin=2` ms and TCP NewReno. The bandwidth of all the links is 1 Gbps. The packet size is 1 KB, and the bottleneck buffer is 128 KB. The propagation delay is about 117 microseconds. We can observe that TCP with GIP outperforms the other two mechanisms. In TCP NewReno, the goodput collapse when

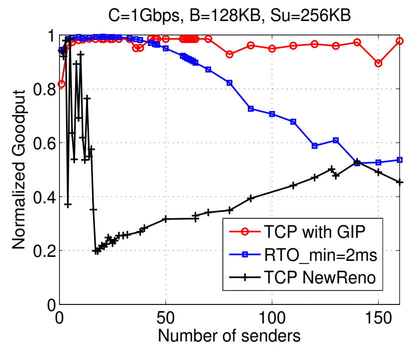


Fig. 15. Normalized Goodput versus the number of senders with C=1Gbps.

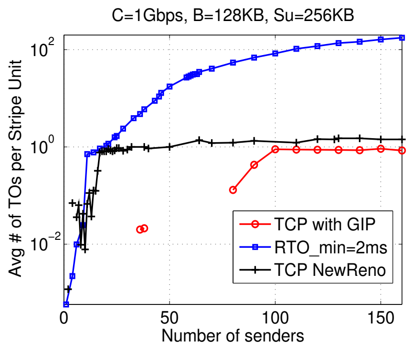


Fig. 16. Average number of TO periods suffered by the unluckiest flow in one stripe unit.

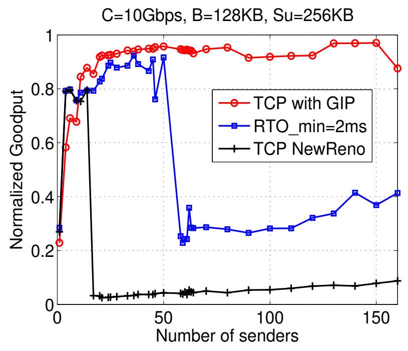


Fig. 17. Normalized Goodput versus the number of senders with C=10Gbps.

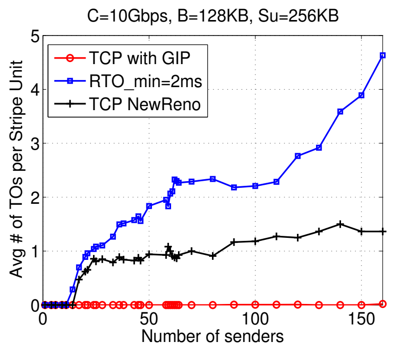


Fig. 18. Average number of TO periods suffered by the most unfortunate flow in one stripe unit with 10 Gbps link.

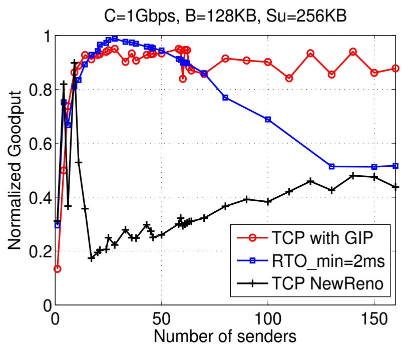


Fig. 19. Normalized goodput versus the number of senders with TCP background traffic.

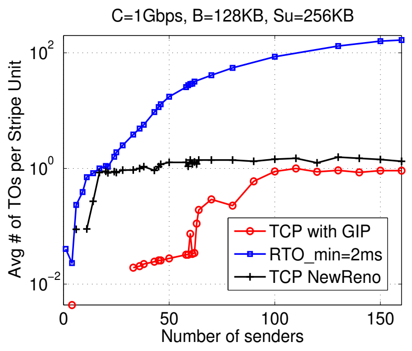


Fig. 20. Average number of TO periods suffered by the most unlucky flow in one stripe unit with TCP background traffic.

the number of sender is larger than about 16. For RTOmin=2 ms, the bottleneck link utilization is larger than 70% when the number of senders is smaller than 100. However, the utilization degrades as the number of senders increases. While TCP with GIP always keeps high goodput.

Fig. 16 exhibits the average number of the TOs suffered by the most unfortunate flow during each stripe unit. TCP with RTOmin=2 ms suffers far more TOs than both TCP NewReno and TCP with GIP do when the number of senders is larger than 50. To present the results more clearly, we use logscale y-axis. Specifically, no marks means that no TO appears. We can see that TCP with GIP suffers few TO periods, which indicates that the FLoss-TOs and LAck-TOs are indeed avoided by reducing cwnd at the start of each stripe unit and redundantly transmitting the last packet for three times at the tail of each stripe unit. The phenomenon that RTOmin=2 ms suffers so many TO periods is caused by the relatively large queue buildup in data centers [13]. As the number of senders increases, the bottleneck buffer is readily overwhelmed. When the buffer is full, the queuing delay at the intermediate switch plus the transmission delay is about $\frac{B}{C} + D = \frac{128\times10^3\times8}{10^9} + 0.117 \approx 1.14ms$. If one packet waits for a relatively long time (i.e., more than 0.86 ms) at the sender or receiver to be processed, the retransmission timer will expire and thus a spurious TO period happens. Therefore, if RTOmin=2 ms, the TOs occur quite frequently. This can also explain why the goodput of RTOmin=2 ms degrades as the number of senders increases in Fig. 15.

Fig. 17 presents the normalized goodput of the three mechanisms with bottleneck bandwidth $C = 10$ Gbps. TCP with GIP outperforms both RTOmin=2 ms and TCP NewReno. Similar to the results in Figure 8, the performance of TCP with GIP is a little worse than both TCP NewReno and TCP with RTOmin=2 ms when the number of senders is quite small, but the link utilization of TCP with GIP is about 70% higher than the other two mechanisms as the number of senders is larger than 50. Compared with Fig. 15, RTOmin=2 ms performs worse with higher bottleneck bandwidth.

Fig. 18 shows the average number of the TO periods suffered by the most unlucky flow with $C = 10$ Gbps. In RTOmin=2 ms, the average number of the TO periods per stripe unit is smaller than that in Figure 16. This is because the queuing delay is smaller when $C = 10$ Gbps, fewer spurious TOs appear. However, the bandwidth wasted during one TO period is larger when the link capacity is higher. Therefore, the normalized goodput is still small.

B. With Background Traffic

The background traffic is not configured in the above simulations. In this subsection, we evaluate the performance of GIP with background traffic. A long term TCP flow is configured as the background traffic. Figure 19 presents the normalized goodput of the TCP with GIP, TCP with RTOmin=2 ms and TCP NewReno. Compared with the results in Figure 15, the goodput of all the three mechanisms decrease slightly. However, the whole tendency does not change. TCP NewReno still suffers goodput collapse as the number of senders increases, RTOmin=2 ms can not work well as the number of senders increases. While TCP with GIP can achieve stable and high goodput. Fig. 20 shows the average number of the TOs suffered by the most unfortunate flow per stripe unit. Compared with the results in Figure 16, the average number of the TOs in the scenario with TCP background traffic is a little more. This is because the bottleneck bandwidth is shared by more flows, and thus more packets are possibly dropped.

VIII. Discussion

A. *If the Number of Senders is Quite Small*

One limitation of TCP with GIP is under-utilization of the bottleneck link when there are only several senders. However, in practice, the number of senders is usually not very small, otherwise the incast collapse will not happen. For example, in PanFS architecture with 64 KB stripe unit size, the throughput collapse does not occur until the number of OSD (Object-based Storage Service) is larger than about 14 [7]. Furthermore, the bottleneck link usually carries background traffic, which reduces the available bandwidth for the incast application.

Next, we compute the size of one stripe unit required to ensure that the TCP with GIP sufficiently utilizes the bottleneck link. Since we reduce the `cwnd` of each sender to 2 at the beginning of each stripe unit. Let A_{ss+ca} denote the total number of packets transmitted during a slow start phase and the following CA phase. If the product of the stripe unit size and the number of senders i exceeds A_{ss+ca}, that is

$$i \times S_u \geq A_{ss+ca} \tag{1}$$

the link will be sufficiently utilized. According to the TCP congestion control mechanism [31], the expected number of the transmitted packets during a slow start phase and the following CA phase, A_{ss+ca}, is

$$\begin{aligned} A_{ss+ca} &= \sum_{i=1}^{\log_2 \frac{W(i)}{2}} 2^i + \sum_{i=0}^{\frac{W(i)}{2}} (\frac{W(i)}{2} + i) \\ &= 2^{\log_2 \frac{W(i)}{2}+1} - 1 + \frac{3}{8}(W(i))^2 + \frac{3}{4}W(i) \end{aligned} \tag{2}$$

where $W(i)$ is the maximum congestion window size of one connection. During a RTT period, the number of the incoming packets is $i \times W(i)$, and the number of the outgoing packets is $C \times RTT = C \times (D + \frac{B}{C}) = CD + B$. According to the flow conservation principle, we can get

$$W(i) = \frac{CD + B}{N} \tag{3}$$

When the number of senders $i = 1$, there is no barrier synchronized transmissions. The application will not suffer the throughput collapse. Hence, in the incast applications, the number of senders at least equals to 2. If C=1 Gbps, D=100 us, B=64 KB, i=2, S_p=1 KB, then $A_{ss+ca} \approx 608$ pkts.

Figure 21 presents the minimum stripe unit size required to saturate the bottleneck link. We can see that, as the number of the senders increases, the minimum stripe unit size decreases dramatically. When the number of the senders is larger than about 6, only less than 10KB stripe unit can fill the bottleneck link. And when $i > 14$, only less than 1KB is enough for TCP with GIP to sufficiently utilize the bottleneck link. What's more, the number of the senders is generally several hundreds or even thousands in the applications with incast communication pattern, such as in the Facebook's memcache system [16] and in the large-scale web query jobs. Therefore, we think that the simplicity and efficiency of TCP with GIP can offset this limitation.

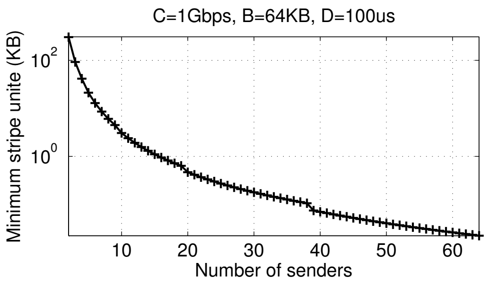


Fig. 21. The minimum stripe unit size required to saturate the bottleneck link without any background traffic.

Furthermore, this limitation of TCP with GIP can be overcome at the cost of increasing complexity. The above under-utilization attributes to the small initial congestion window size. which is similar to the problem described in RFC 3390 [32], which suggests increasing the initial congestion window size to improve the TCP throughput in networks with high bandwidth and large propagation delay. In TCP with GIP, the initial congestion control window size can be increased at the beginning of each stripe unit to improve the link utilization. For example, each flow can set the initial congestion window W_s to $\frac{S_u}{T}$, where S_u is the stripe unit size in units of packets. T is the time spent transmitting the last stripe unit. S_u and T can be estimated. However, the goal of TCP with GIP is to solve the TCP incast problem as the number of the senders is relatively large by modifying TCP as less as possible, and this problem has little impact on TCP with GIP. Thus, the dynamic initial window mechanism is not investigated in depth in this work.

B. *Feasibility of GIP*

The main difference of the incast applications and other TCP-based applications is the *barrier traffic pattern*. The GIP mechanism shuns the impact of the pattern through some specific process, namely, reducing `cwnd` at the start of each stripe unit and redundantly transmitting the last packet of each stripe unit for at most 3 times. Next we will discuss whether this enhanced mechanism affects TCP's normal running.

In the Linux TCP implementation, packets can not be sent until `cwnd > tcp_packets_in_flight()`. After successful transmission of one stripe unit, the value of `tcp_packets_in_flight()` will be cleared to 0. Hence, reducing `cwnd` to 2 at the start of each stripe unit doesn't affect the normal work of TCP.

The second part of GIP is redundantly transmitting the last packet of each stripe unit for 3 times using `tcp_retransmit_skb()` function. Since each redundant packet can not be sent until one ACK is received, it obeys the pipeline model of TCP. If some of the last three packets in a stripe unit are lost, and the three redundant packets are successfully transmitted, then at least 3 duplicate ACKs are sent to the sender to trigger FR/FR procedure. Otherwise, if no packet of one stripe unit is lost, the receiver will receive the last packet for 4 times. Hence it responses to the sender with 3 duplicate ACKs. But this will not trigger unnecessary FR/FR procedure, since when all the data is transmitted successfully, both the transmitting and the retransmitting queue at the sender are empty and all the socket buffers have been released. Only the value of `tp->retrans_out` will be influenced by GIP. After `tcp_retransmit_skb()` is called for 3 times, `tp->retrans_out` will increase by 3. Even if 3 duplicate ACKs for the last packet are received, `tp->retrans_out` will not decrease since the last packet has been successfully transmitted, which means the retransmission is unnecessary. As a result, `tp->retrans_out` will increase by 3 after the transmission of each stripe unit, which makes `tcp_packets_in_flight()` also increase by 3. After several units are transmitted, `tcp_packes_in_flight()` will be larger than `cwnd`, which blocks the packet transmissions. In normal state, `tp->retrans_out` decreases to 0 after one stripe unit is finished. Hence, GIP resets `tp->retrans_out` to zero to ensure TCP work normally.

IX. Conclusion

In this paper, an enhanced mechanism, GIP, is designed and implemented to solve the TCP incast problem. To avoid FLoss-TOs and LAck-TOs, which are the main causes of the TCP incast problem, the GIP mechanism reduces the congestion window at the start of each stripe unit and redundantly transmits the last packet of every stripe unit for at most three times. GIP is implemented in a testbed with 24 servers and Ethernet Gigabit Switches. The experimental results validate that GIP can properly overcome the incast problem at low cost. Furthermore, we conduct series of simulations on the ns-2 platform to evaluate the performance of GIP with higher bottleneck bandwidth and larger number of senders. The simulation results demonstrate it has good scalability.

Acknowledgement

The authors gratefully acknowledge the anonymous reviewers for their constructive comments. This work is supported in part by the National Natural Science Foundation of China (NSFC) under Grant No. 61225011, National Basic Research Program of China (973 Program) under Grant No.2012CB315803 and 2009CB320504.

References

[1] V. Vasudevan, A. Phanishayee, H. Shah, E. Krevat, D. G. Andersen, G. R. Ganger, G. A. Gibson1, and B. Mueller, "Safe and Effective Fine-grained TCP Retransmissions for Datacenter Communication," in *ACM SIGCOMM*, pp. 303–314, Aug.2009.

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

[3] Y. Chen, R. Griffith, J. Liu, R. H. Katz, and A. D.Joseph, "Understanding TCP Incast Throughput Collapse in Datacenter Networks," in *ACM WREN*, 2009.

[4] F. Chang, J. Dean, S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber, "Bigtable: A Distributed Storage System for Structured Data," *ACM Transactions on Computer Systems (TOCS)*, vol. 26, no. 2, p. 4, 2008.

[5] "Apache HBase," in *http://hbase.apache.org/*.

[6] "Hypertable-project home page," in *http://hypertable.org/*.

[7] D. Nagle, D. Serenyi, and A. Matthews, "The Panasas ActiveScale Storage Cluster: Delivering Scalable High Bandwidth Storage," in *ACM/IEEE Supercomputing*, pp. 53–62, 2004.

[8] R. E. Bryant, "Data-Intensive Scalable Computing: Harnessing the Power of Cloud Computing," 2009.

[9] J. Dean, S. Ghemawat, and G. Inc, "MapReduce: Simplified Data Processing on Large Clusters," in *USENIX OSDI*, 2004.

[10] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed Data-parallel Programs from Sequential Building Blocks.," in *EuroSys*, pp. 59–72, 2007.

[11] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster Computing with Working Sets," in *USENIX HotCloud*, 2010.

[12] P. Costa, A. Donnelly, A. Rowstron, and G. O'Shea, "Camdoop: Exploiting In-network Aggregation for Big Data Applications," in *USENIX NSDI*, 2012.

[13] M. Alizadeh, A. Greenberg, D. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, M. Sridharan, and C. Faster, "DCTCP: Efficient Packet Transport for the Commoditized Data Center," in *ACM SIGCOMM*, pp. 63–74, 2010.

[14] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowstron, "Better Never than Late: Meeting Deadlines in Datacenter Networks," in *ACM SIGCOMM*, pp. 50–61, 2011.

[15] C. Hong, M. Caesar, and P. Godfrey, "Finishing Flows Quickly with Preemptive Scheduling," in *ACM SIGCOMM*, 2012.

[16] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, *et al.*, "Scaling Memcache at Facebook," in *USENIX NSDI*, pp. 385–398, 2013.

[17] H. Wu, Z. Feng, C. Guo, and Y. Zhang, "ICTCP: Incast Congestion Control for TCP in Data Center Networks," in *ACM CoNext*, 2010.

[18] "IEEE P802.3ba 40Gb/s and 100Gb/s Ethernet Task Force," in *http://www.ieee802.org/3/ba/index.html*.

[19] J. Duffy, "100G Ethernet Coming to a Cisco Rack Near You," in *http://www.networkworld.com/community/blog /100g-ethernet-coming-cisco-rack-near-you*, 2011.

[20] M. Allman, H. Balakrishnan, and S. Floyd, "Enhancing TCP's Loss Recovery Using Limited Transmit," in *http://www.faqs.org/rfcs/rfc3042.html*, 2001.

[21] A. Romanow and S. Floyd, "Dynamics of TCP Traffic over ATM Networks," *ACM SIGCOMM CCR*, vol. 24, no. 4, pp. 79–88, 1994.

[22] P. Devkota and A. Reddy, "Performance of Quantized Congestion Notification in TCP Incast Scenarios of Data Centers," in *MASCOTS*, pp. 235–243, 2010.

[23] Y. Zhang and N. Ansari, "On Mitigating TCP Incast in Data Center Networks," in *IEEE INFOCOM*, pp. 51–55, 2011.

[24] J. Zhang, F. Ren, and C. Lin, "Modeling and Understanding TCP Incast in Data Center Networks," in *IEEE INFOCOM*, pp. 1377–1385, 2011.

[25] D. Clark and D. Tennenhouse, "Architectural Considerations for a New Generation of Protocols," *ACM SIGCOMM CCR*, vol. 20, no. 4, pp. 200–208, 1990.

[26] S. Shenker, L. Zhang, and D.D.Clark, "Some Observations on the Dynamics of a Congestion Cotrol Algorithm," *ACM SIGCOMM CCR*, pp. 30–39, Oct., 1990.

[27] A. Medina, M. Allman, and S. Floyd, "Measuring the Evolution of Transport Protocols in the Internet," *ACM SIGCOMM CCR*, vol. 35, no. 2, pp. 51–55, Apr. 2005.

[28] R. Braden, "Requirements for Internet Hosts – Communication Layers," Internet Engineering Task Force, RFC 1122.

[29] V. Vasudevan, "Linux TCP Microsecond Timers Patch," in *https://github.com/vrv/linux-microsecondrto*, 2010.

[30] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica, "Managing Data Transfers in Computer Clusters with Orchestra," in *ACM SIGCOMM*, 2011.

[31] W. Stevens, "TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms," RFC 2001, 1997.

[32] M. Allman, "Increasing TCP's Initial Window," RFC 3390, Oct. 2002.