Preventing TCP Incast Throughput Collapse at the Initiation, Continuation, and Termination

Adrian S.-W. Tam Kang Xi Yang Xu H. Jonathan Chao
Department of Electrical and Computer Engineering
Polytechnic Institute of New York University
Email: adriantam@nyu.edu, kxi@poly.edu, yangxu@poly.edu, chao@poly.edu

Abstract*—Incast applications have grown in popularity with the advancement of data center technology. It is found that the TCP incast may suffer from the throughput collapse problem, as a consequence of TCP retransmission timeouts when the bottleneck buffer is overwhelmed and causes the packet losses. This is critical to the Quality of Service of cloud computing applications. While some previous literature has proposed solutions, we still see the problem not completely solved. In this paper, we investigate the three root causes for the poor performance of TCP incast flows and propose three solutions, one for each at the beginning, the middle and the end of a TCP connection. The three solutions are: admission control to TCP flows so that the flow population would not exceed the network's capacity; retransmission based on timestamp to detect loss of retransmitted packets; and reiterated FIN packets to keep the TCP connection active until the the termination of a session is acknowledged. The orchestration of these solutions prevents the throughput collapse. The main idea of these solutions is to ensure all the on-going TCP incast flows can maintain the self-clocking, thus eliminates the need to resort to retransmission timeout for recovery. We evaluate these solutions and find them work well in preventing the retransmission timeout of TCP incast flows, hence also preventing the throughput collapse.

I. Introduction

The following is an example of a scenario of incast: A client (a user application) reads a file stored in a parallel network file system (pNFS [1], [2]). The file in pNFS has multiple data blocks. Each block is stripped and stored at different locations over the network for a higher read/write throughput. After the client learned about the locations of the stripped data blocks of the file, it connects to the storage nodes over TCP, e.g., through iSCSI protocol, to retrieve the data blocks and reassemble them locally, as illustrated in Fig. 1. This retrieve-reassemble process repeats for all data blocks of the file.

Several characteristics of incast can be seen above. Firstly, the return path of the data from the data sender (storage) to the aggregator (pNFS client) is a high fan-in *many-to-one* communication. Secondly, the data retrieved from each storage location, known as the server request unit (SRU) is of a *small size*, typically in several KB and seldom larger than a hundred. Thus, the data connection is usually *short-lived*. Thirdly, the aggregator usually processes the data sequentially. While the requests are sent to the data senders at the same time, the next round of requests cannot start until all senders in the previous round have finished sending. In parallel computing terms, it is *barrier synchronized*. Finally, this type of application is executed in a fast network with *small buffer* so as to have faster round-trip times by short queueing delay.

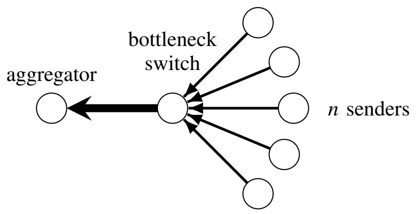

Fig. 1: Data flow of incast is from many senders to a single data aggregator

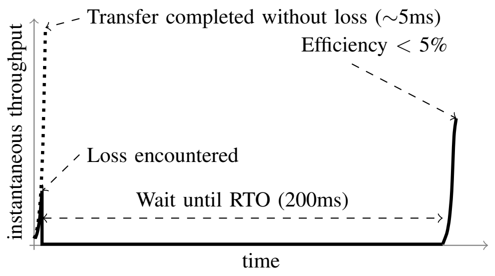

Fig. 2: Goodput degrades if a short-lived flow has a RTO

These characteristics coincidently cause the throughput collapse in many incast applications: The many-to-one traffic creates an imbalanced data rate between the ingress and egress ports at the bottleneck switch, which exhausts the buffer quickly. Consequently, there is a massive loss of packets, which is so severe that TCP cannot recover it by fast retransmission algorithm. Thus, retransmission timeout (RTO) occurs, with the duration no less than 1 second according to RFC2988 or no less than 200ms in many operating systems [3]. In the meantime, no data is delivered. Compare to the round-trip time (RTT) within a millisecond, and the lifetime of a typical incast flow of a few milliseconds [4], a minimum RTO (RTO_{min}) in hundreds of milliseconds is too long. Whenever a RTO occurs during incast, the transfer is severely lengthened (see Fig. 2). Because of the barrier synchronized nature of the application, if such delay happens in every round of requests, the degraded performance is very noticeable.

Incast throughput collapse is not only observable in storage networks [5], [2] but also in MapReduce [6] or other partition-

978-1-4673-1298-1/12/$31.00 © 2012 IEEE

aggregate applications that run on data center networks [7]. A literature review is presented in section VI. As the incast throughput collapse is caused by RTO, it is trivial to solve the problem by reducing the RTO duration [4] for 2–3 orders of magnitude to a microsecond level, but there are technical difficulties due to time granularity in operating systems or will increase the operation overhead significantly due to more frequent timer interrupts.

In this paper we ask the fundamental question of what causes the TCP to have RTO. We find that, despite it is time-tested, TCP is not robust enough to discard RTO entirely. In section II, we break down the immediate causes of RTO into three different cases: block loss, double loss, and tail loss. To attack each of these three cases, we propose three solutions in section III, namely, the admission control to TCP flows, timestamp-assisted retransmission, and reiterated FIN packets. They together present a complete solution to the incast throughput collapse problem. In the evaluations in section IV, we find them working well. We also discuss about these solutions in section V before we conclude the paper.

II. The Three Causes of RTO

TCP, NewReno[*] in particular, recovers lost data by the fast retransmit algorithm: three duplicate acknowledgments (dupacks) arriving at the data sender is a signal that conveys the corresponding data packet is lost and triggers retransmission immediately. This retransmission is before the RTO to expire, thus called fast retransmit. After the retransmission, the sender is expecting a new acknowledgment (ack). If there is any loss that fast retransmit cannot recover (as shown below), TCP resorts to RTO for recovery.

Fast retransmit does not work well in some situations: Acks and dupacks are generated at the receiver only when there is a packet arrival. If the TCP sliding window can hold n packets, loss of any n consecutive packets makes no acknowledgment can be generated. It will leave no *in-flight* packets on the network and break the TCP *self-clocking*, as shown in Fig. 3a. We call this situation the *block loss*. Note, if RFC3042 limited transmit [8] is not in effect, losing $n-2$ packets among any n consecutive packets creates such deadlock.

Another situation is loss of the retransmitted packet, we name this *double loss* on the same data. After fast retransmit, the TCP sender is expecting a new ack. Any dupack received further is assumed to be corresponding to data sent before the fast retransmit. If the sliding window allows, these further dupacks trigger the transmission of *new* data segments only, whereas the retransmitted packet will not be retransmitted again until RTO, as illustrated in Fig. 3b. In other words, TCP assumes the retransmitted packet must arrive at the receiver.

The last situation that fast recovery fails is due to not sending enough packets to trigger triple dupacks. As illustrated in Fig. 3c, at the last few packets of a TCP stream, if we do not have enough data to send after the lost packet, the receiver will not generate enough dupacks to trigger fast retransmit. Then, the loss is recovered by RTO only. We call this the *tail loss*.

These three situations correspond to different phases of a TCP connection. Block loss occurs most easily at the early phase of a connection when the window is small during the slow-start period. Double loss happens in the middle of the connection, after the first packet loss is encountered. Tail loss, by definition, happens only at the last few packets of the TCP stream. In [9], it is argued that the relative likelihood of block loss and tail loss is related to the size of SRU. Therefore, they are equally important in causing RTO when we consider the wide spectrum of incast traffic pattern.

To *prevent* incast throughput collapse, we must prevent all these three cases of loss to occur. While some previous work [10], [9] also mention some of these causes for RTO, to the best of our knowledge, we are the first to address all of them to solve the incast problem.

III. The Three Solutions

As seen in section II, we cannot dispose RTO in TCP because we need it quite often to recover the lost packets. However, we can improve TCP to make it less depending on RTO. We propose three different solutions, each to tackle one cause of the RTO mentioned in section II. Our objective is to prevent RTO from happening as much as possible, so that the incast throughput collapse is consequently prevented.

The following solutions are abided by several design rules. Firstly, they are event-driven solutions that do not use timers. This is not only because timers are scarce resources in an operating system, but also their granularities are usually too coarse to be useful in data center networks. Secondly, we do not introduce new protocol headers but to reuse the existing elements in the TCP/IP headers as much as possible. Therefore, in some of the following solutions, we demonstrate TCP is not fully utilizing its capability for loss recovery. Thirdly, we keep the design simple. We do not manipulate the congestion control algorithm to avoid instability of networks.

A. Admission control as a solution to block loss

In an incast application, throughput collapse is inevitable given enough number of concurrent senders. This can be understood in this simple example: Assume a bottleneck buffer can hold at most k packets. If there are more than k senders, each sender can have a share of less than one packet's size in the buffer. Thus, packet drop is inevitable. With even more senders, however slow they send, the statistical multiplexing must fail eventually and the packet drop would be massive, leading to RTO due to block loss.

Apparently, we must limit the number of concurrent senders in the incast application to prevent incast throughput collapse, an idea united with [11], [12], [13]. However, the incast problem is caused by the interaction between the buffer sizes, the senders, and their synchronized traffic pattern. Determining the optimal number of concurrent senders must take the network situation into account.

[*]While we focused on TCP NewReno (RFC2582), the discussion in this paper is valid for all other modern TCP variants because those variants differ only on the congestion control mechanism. Their flow control, i.e. loss detection and recovery procedure, remains the same.

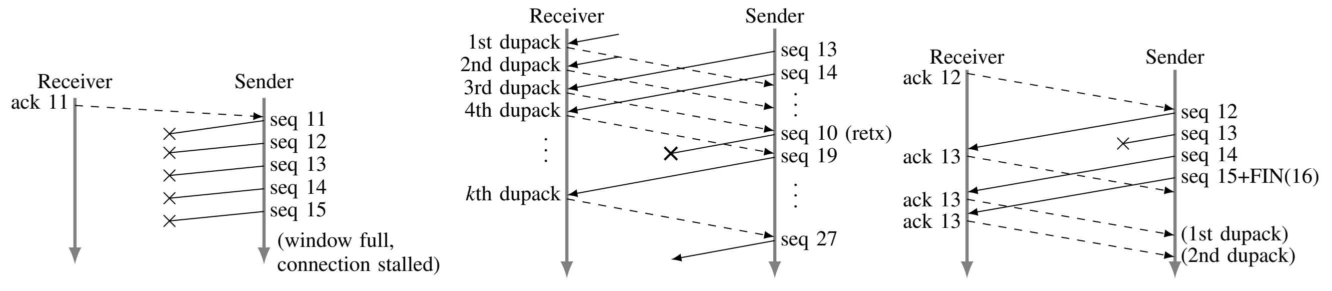

Fig. 3: Three causes of RTO: (a) block loss, packets from the whole window are lost; (b) double loss: the retransmitted packet is lost again; and (c) tail loss, one of the last few packets from the stream is lost.

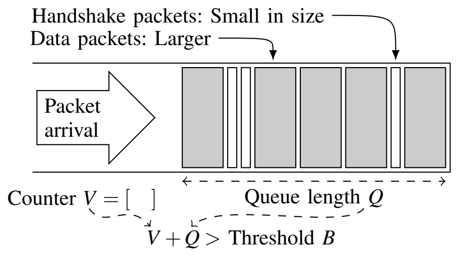

Fig. 4: Admission control of incast flows

We propose admission control to incast TCP flows by the cooperation between the incast aggregator and the bottleneck switch. The bottleneck switch reports to the aggregator about the forecasted buffer usage, and the aggregator withholds certain TCP flows accordingly.

This is a solution to block loss: The probability of block loss is high when the TCP has a small window, such as the time of slow start at its early phase, when the congestion window is inflating from a small size. Packet drops at such time could be catastrophic. Therefore, we have to make room for each flow to inflate its window, by limiting the total number of concurrent incast flows, so that the probability of block loss can be reduced. Details of the admission control are as follows.

Prediction of buffer usage using three-way handshake: TCP performs three-way handshake to establish a connection before it sends any data. In the three-way handshake, illustrated in Fig. 5, each end of the connection sends a SYN-flag-carrying packet to exchange the initial sequence numbers. Therefore, the intermediate switch between the two ends must see a SYN flag traversing in both directions before the data is sent.

When we consider the size of packets, there is a vast difference between the handshake packets and the data packets. The MTU of Ethernet is 1500 bytes, which is the usual packet size for a TCP data packet. Signaling packet, such as TCP's acknowledgment packets and handshake packets, contains only the headers and its total size is normally 40–60 bytes. Taking this fact, we devise the following prediction scheme.

In the switch, output-buffering assumed, we install a counter to each output port. When a packet is being queued at an output port, and if that packet is a TCP packet with the SYN flag asserted, the switch increases the counter by N, as illustrated in Fig. 4. This increment is to reflect that, at one RTT later, the hosts in concern have the connection established and sending data. By then, the switch buffer shall be consumed by N more bytes. Therefore, we name the counter as *virtual byte counter*. The virtual byte counter will be decremented, at a rate same as the line speed, only when the output port is idle and the corresponding buffer is empty. As if there were that many bytes in the buffer, by that time, they shall be depleted at the line speed. The initial size of the congestion window in an established TCP connection is one to four segments [14], and thus we set N to four times the MTU.

The switch predicts the buffer utilization by the sum of the current buffer usage Q and the value in the virtual byte counter V. If $Q+V$ is greater than the size of buffer B, it is expected to see a buffer overflow in an RTT, when the TCP connections are established. Thus, the switch would mark the incoming SYN-flag-carrying packets if this condition is met. The marking facility is readily available in the IP header: the two-bit ECN field [15]. This is also the standard way to convey congestion information to the hosts when RED queues are used [16]. But in this case, we concern about the TCP handshake packets only, and to convey a predicted congestion rather than a confirmed congestion.

Delay of connections: The actual admission control is performed at the incast aggregator. This makes the non-incast application transparent to the changes, since ECN-marked TCP handshake packets are ignored in other applications [15].

The incast aggregator can infer that buffer overflow is predicted when it sees the marked handshake packets. Therefore, those TCP connections shall not be established at once. We devised two methods, as follows, to delay the connections.

The first method is to withhold a handshake, as no TCP can send any data before the three-way handshake is completed. Refer to Fig. 5, if the aggregator is undergoing passive open [17], and the incoming SYN packet is marked, it can withhold the SYN+ACK packet so that the connection can be suspended. Similarly, if the aggregator is undergoing active open, and the incoming SYN+ACK packet is marked, it can withhold the ACK packet that completes the three-way handshake to suspend the connection. The withheld packet will be sent later

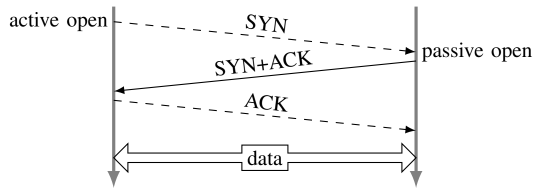

Fig. 5: Three-way handshake

to resume the connection. Implementing this can be done by modifying the BSD socket API at the aggregator so that, by setting the incast-specific options in `setsockopt()` call, the arrival of a marked handshake packet can notify the application for the suspension-resumption handling.

The second method, which involve less change in the socket API, is to suspend a connection by announcing a receive window of zero size. The window field in TCP header (the 15th–16th byte) specifies how much data the receiver can accept at once. Setting this to zero can effectively stop the sender from sending data. Thus, we can suspend a connection by putting a zero window in the responding handshake packet if the incoming one is marked. To resume the connection, the aggregator simply spontaneously sends an ack packet to the sender with the inflated window field.

In both methods, the aggregator remembers all the connections, both the established and the suspended, to make the sender staggering automatic: All the senders can attempt to connect to the aggregator, but as some of them would be suspended, only a limited number of connections are sending data at any time. When one of these data-sending connections completed, the aggregator can resume one of those suspended connections, so that we can prevent an overwhelming number of TCP flows using the network simultaneously.

This suspension-resumption design is suitable for incast applications as their flows are short-lived by nature. Otherwise, the overwhelmed bottleneck buffer shall be alleviated by the congestion control algorithm instead, and RTO is tolerable due to the long holding time of flows. The short-lived nature of incast flows also implies that the duration of withhold is short, in the order of milliseconds. TCP connection retry will not take place in such short delay[†]. Hence one can hardly tell from the network whether there is a suspension taken place.

Section V discusses some further issues about the design.

B. Timestamp-assisted retransmission to recover double loss

Identifying double loss is impossible in TCP per se. It is because we cannot tell if an incoming ack packet corresponds to the data that was sent before or after a retransmission, hence we do not have a definite time to reset the dupack count for a second retransmission of the same data packet. Resolving to RTO in this scenario is explicitly stated in [18].

Fortunately, the timestamp option for TCP, specified in [19] can help. This is created to prevent the sequence number wrap-around problem, and all modern OS use this by default due

[†]In Linux, for example, first connection retry takes place at 3 seconds later if no response is received during the three-way handshake.

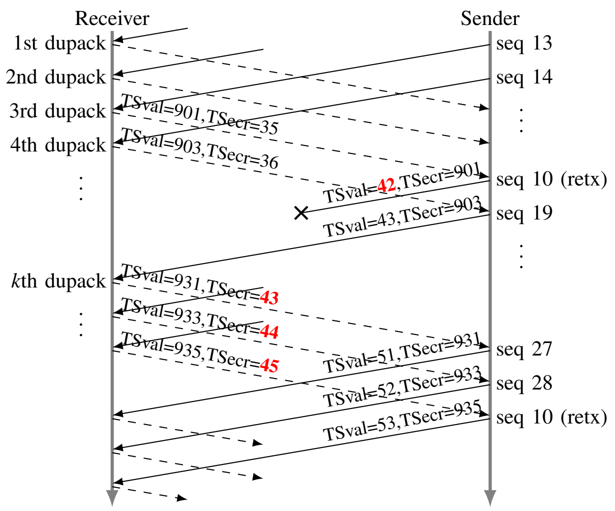

Fig. 6: Use of timestamp-assisted retransmission to recover double loss, mimicking the triple dupack trigger. T_{retx} in this example is 42.

to its cruciality to guarantee accurate delivery in high speed networks. It is also used by the OS to estimate RTT. We extend its use to identify double loss.

The timestamp option in TCP header carries two timestamp values, the local timestamp (TSval) and the echoed timestamp (TSecr). TSval in a packet corresponds to the time of the local clock source when the packet is sent, and the TSecr is the copy of the latest timestamp a host receives from its TCP peer. Trivially, TSval is always a monotonically increasing function.

Taking this property, we can identify the loss of a retransmitted packet: We record the current timestamp as T_{retx} when we retransmit a packet. Afterwards, we check if any arriving packet carries a timestamp with TSecr $> T_{\text{retx}}$ but does not acknowledge the receipt of the retransmitted packet. This means a packet that was sent after the retransmission arrived first. Then, we can assume the retransmitted packet is lost and retransmit it again if we assume no reordering in the network, or, to mimic the triple dupack trigger, wait for three such arrivals before retransmitting it again. Fig. 6 shows an example of this case, in contrast to Fig. 3b. Since TCP is retransmitting at most one packet at any time, we need just one extra variable per connection to hold T_{retx}.

C. Reiterated FIN packets as a solution to tail loss

The tail loss is unrecoverable by fast retransmit because of not enough data arriving the receiver to trigger triple dupacks. There would not be any solution unless we can append dummy bytes to lengthen the TCP data stream. Fortunately, such 'dummy bytes' exist.

In order to protect important signals in TCP, sequence numbers are assigned to two special flags in TCP header, namely, the SYN and FIN flags, which signals for connection establishment and termination respectively. Thus, an empty TCP packet with the FIN flag asserted in the header is as if it is carrying one byte of data, and shall be acknowledged upon

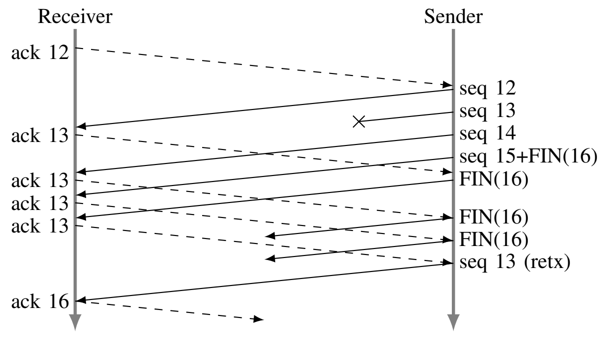

Fig. 7: Reiterated FIN packets to recover tail loss

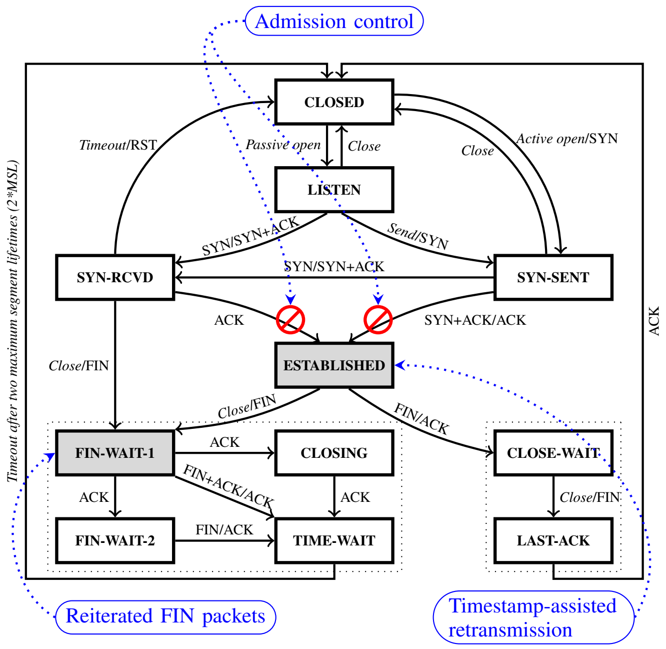

Fig. 8: The three solutions to prevent RTO

receipt. If one of the last three packets of a data stream is lost, but some empty FIN packets arrive subsequently, there will be enough dupacks to trigger the retransmission.

Hence, we propose to resend the FIN packet until the FIN packet itself is acknowledged, which assures the receipt of all user data. After the first FIN packet is sent, the sender's TCP transits from ESTABLISHED state to FIN-WAIT-1 state [17]. Then, any arriving ack packet that is not acknowledging the FIN triggers the reiterated transmission of a FIN packet carrying no data, as illustrated in Fig. 7 (in contrast to Fig. 3c). When an ack packet that acknowledges the FIN flag is received, the TCP sender proceeds to FIN-WAIT-2 or TIME-WAIT state and stopped reiterating FIN packets. Since the reiterated FIN packets carry no data, their tiny size give minimal impact to the network load.

Summary of the three solutions introduced in this section is depicted in the TCP state diagram in Fig. 8.

IV. Evaluation

A. Analysis of the probability of RTO

We first present a qualitative analysis of the RTO probability, to show how the three solutions proposed in section III can help. A more thorough, detailed analysis is part of our future work. Since we do not consider the correlated loss probability between packets, the result below shall not be interpreted quantitatively.

For simplicity, we assume the same loss probability p for each packet in the network. Due to the imbalance of traffic intensity in incast applications, we further assume the acknowledgments from incast aggregator to incast senders is lossless. Moreover, to simplify calculation, we assume limited transmit (RFC3042 [8]) is enabled. Without which can only increase the probability of RTO in the calculations below, as the TCP is easier to lose its self-clocking.

1) Admission control and block Loss: Block loss is to lose all the packets in a window, i.e., all the n packets in a row, which has the probability of p^n. One can easily see that, the probability decreases exponentially with the window size n.

With a large transmission window, block loss is unlikely. But its probability becomes significant if the window is small. Thus, when the TCP congestion window is inflating from a small size during slow-start at its early phase of the transfer, a packet drop could be catastrophic. Admission control is to prevent this to happen by avoiding severe congestion during the first few RTT of the connection. With admission control, the number of concurrent incast senders is limited. This also provides more room for the other solutions to operate.

2) Timestamp-assisted retransmission and double loss: We just need to lose both the original and retransmitted packets to make a double loss. The probability of such is p^2.

If the timestamp-assisted retransmission is enabled, RTO occurs only when all the clues are lost, i.e., the retransmitted packet and all the packets sent after T_{retx}. During fast recovery, a new data packet is sent for every additional dupack or partial acks received. So the total data packet that could be sent on and after T_{retx} is equal to the window size n. So the probability of RTO is equal to p^{n+1}, a significant decrease from p^2.

3) Reiterated FIN packets and tail loss: Tail loss happens when any of the last three packets in the TCP stream is lost, which the last one is the one to carry the FIN flag normally. Hence it has the probability $1-(1-p)^3$.

Assume TCP sender's window is of the size of n packets at the time that the first FIN packet is sent. With reiterated FIN, tail loss is to lose any of the last three packets in the stream *and* all the reiterated FIN packets, where there are $n-k$ of them, with $k=1,2,3$ denotes the number of packets among the last three is lost. So the probability reduces to $\sum_{k=1}^{3} p^{n-k}\binom{3}{k}(1-p)^{3-k}p^k = p^n[(2-p)^3-(1-p)^3]$.

From the above, it is obvious that RTO is less likely to happen when we have our solutions in effect.

B. Simulation

We implemented the three solutions in section III, given the acronym *ICaT* to stand for the initiation, continuation

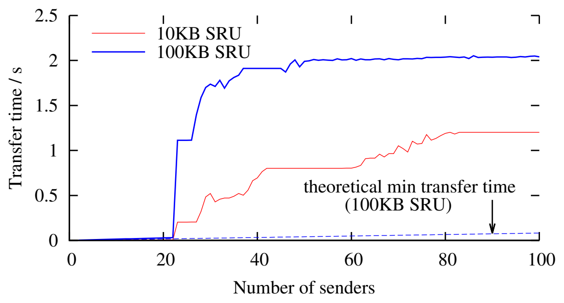

Fig. 9: Transfer times using vanilla TCP. The dotted line shows the theoretical minimum bound.

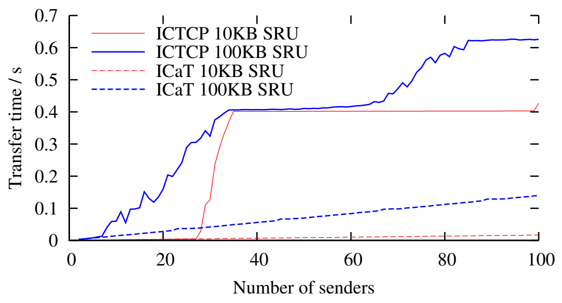

Fig. 10: Transfer time using ICTCP and ICaT

and termination, in NS-3 [20]. We set the network with a fast round-trip propagation delay of 100μs and each link has the speed of 1Gbps. The total end-to-end processing delay is uniformly distributed in the range of 0–$15\mu s$[‡] to reflect the reality [21], [22]. There is a single aggregator and n incast senders, $1 \leq n \leq 100$, connected by an output-buffered bottleneck switch with 128KB buffer per port[§], as in Fig. 1. RTO_{min} is set to 200ms, the standard value in most OS nowadays. Since the RTT is short, at most 1ms with queueing delay accounted, the actual retransmission timeout in the simulation will be RTO_{min}. All data mentioned below are the average results of 100 rounds of simulation.

We first demonstrate the behavior of incast when a standard (a.k.a. vanilla) TCP is used. Fig. 9 shows the time spent for transmitting one SRU against the number of incast senders. Two different sizes of SRU, namely, 10KB and 100KB are shown. The transfer time for the incast is the time that all the transfer completed. In other words, it is the transfer time of the slowest sender.

Generally, the curve for 100KB SRU is above that of 10KB SRU, as the more data to send, the longer it should take. Before the number of senders reaching 20, the transfer time is within a few milliseconds. But once the number of senders get larger, the transfer time increased dramatically. Both curves start to have a significant increase of transfer time at the same number of senders. This proves that the throughput collapse due to overwhelming number of senders is independent of the SRU. By examining detailed packet trace, we find that some flows experience several RTOs in its lifetime, which makes the transfer time roughly at multiples of 200ms. But as the number of senders increased even further, such as more than 60, we observe a different RTO value: We start to see flows to have block drop at the very first data packet of its transfer. At such time, since there is no effective RTT samples collected at the TCP, the default RTO value is one second. Thus we can see the transfer time in terms of seconds at that range. Note that, since we are considering the total transfer time or the time of the slowest flow, increasing the total number of sender does not necessarily make the transfer longer. This explains the leveling curve in Fig. 9.

For 100 senders with 100KB SRU, the total amount of data is around 10MB, which a 1 Gbps link can send it in 81ms. Any transfer time of several hundred milliseconds can be regarded as an evidence of throughput collapse. The result in Fig. 9 proves our claim in section III that given enough number of concurrent senders, throughput collapse is inevitable.

Next we show the effectiveness of our solutions in section III. Among the three solutions, timestamp-assisted retransmission and reiterated FIN are modifying the established TCP connections to maintain the self-clocking, i.e., to guarantee there is always a packet in transit to keep the connection active. Admission control, however, is preventing an overwhelming number of connections to proceed which will definitely break the self-clocking. All these are *preventive* measure to avoid incast throughput collapse.

We notice ICTCP [24] also position itself as a preventive solution of TCP incast throughput collapse. It is a *cross-socket control* in the sense that information is gathered from all the established TCP sockets and their sending rate is tuned based on such information. It measures the sockets' incoming data rate periodically, and then adjusts the rate by modifying the announced receive window. Due to its similar nature, we compare our solutions with ICTCP. The result is shown in Fig. 11 and 10. Note, our simulation is different from the evaluation in [24] in two ways: Firstly, the buffer size in our simulation is smaller. This can magnify the problem of incast throughput collapse. Secondly, the size of SRU is smaller. We believe such SRU size is closer to the reality. Due to such differences, it is easier to see the limitations of ICTCP: It cannot prevent collapse if there are too many senders, and its flow rate adaptation is not immediate. Our solutions, however, can handle a large number of senders because of the admission control scheme, and we do not need to use rate adaptation to prevent packet loss. Indeed, packet loss does not always yield to RTO. Therefore, we do not prevent packet loss but to promptly recover the packet loss using all the possible clues.

[‡]The processing delay of 15μs is actually underestimated. Performing `ping localhost` on a decent computer may report a response time as much as 30μs, which can be reasoned as the host OS's response time to network events, not yet taken into account the network equipment's processing time. This response time, however, could be significantly shorter if TCP/IP offload engines (TOE) are used at the hosts.

[§]Commercial-grade commodity switches usually has more than this amount of buffer capacity per port. For example, Cisco Catalyst 4948 [23] has 16MB buffer for its 48 ports. OpenFlow switches such as the one used in [24] would have even more. Thus the simulation set up is harsher than reality.

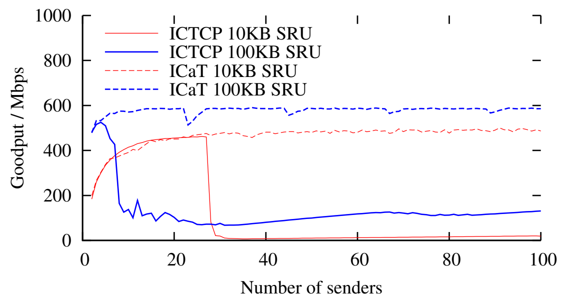

Fig. 11: Goodput using ICTCP and ICaT

Comparing Fig. 9 and 10, trivially, both our solutions and ICTCP improve the incast performance over the vanilla TCP. Comparing ICTCP and our solutions (denoted with ICaT, for the three solutions in section III), we see a steady, linear curve in our solutions in Fig. 10. This reflects the increased total volume of data as the number of senders increase. It shows a roughly constant transmission rate when our solutions are used, regardless the total number of users. ICTCP, however, shows abrupt increases of transfer time. This is due to RTO occurred, even cross-socket congestion control is used.

The degraded performance on vast amount of senders in ICTCP is more obvious when we consider the goodput in Fig. 11. Our solutions show a steady goodput over different number of senders. The larger the SRU, the higher the goodput. This is due to the amortization of the protocol overhead. However, for ICTCP, the goodput decreases significantly once the number of senders becomes large.

Although it is not shown in this paper, we did some more simulations and found that ICTCP is effective to prevent incast traffic from overwhelming the network in different bottleneck buffer sizes or SRU. But only if the total number of users is not too large. This is expected, because ICTCP adjusts the sending rate of each flow based on an estimate. Any error in the adjustment would be magnified when we have a large population of senders. This again confirms our argument that admission control is crucial.

C. Orchestration of the solutions

The three solutions in section III are independent of each other and they could be used individually. Although our simulations confirm that, only their orchestration can effectively avoid RTO in most circumstances, we repeated the simulation on 100 incast senders, with one of each of the three solutions disabled, and counted the number of timeout events. The result is shown in TABLE I, which the numbers show the percentage of flows that see any RTO, and the average number of RTO that they encounter in a flows' lifetime. For example, for a SRU of 10KB, performing admission control and timestamp-assisted retransmission but not reiterated FIN packets will make 15.4% of flow to encounter RTO in its lifetime. And since each flow can see only one tail loss, each of those 15.4% flows sees exactly one RTO. Similar to the simulations above, these numbers are average of 100 rounds.

TABLE I: Per flow RTO count / Percentage of flow encountered RTO

	No adm. control	No timestamp	No re. FIN
10KB SRU	2.1 / 77.2%	1.0 / 15.2%	1.0 / 15.4%
100KB SRU	2.1 / 100%	3.7 / 67.0%	1.0 / 0.2%

On a high number of incast senders, there is a significant performance penalty if we do not perform admission control. For example, on SRU of 100KB, all flows see at least one RTO in their lifetime. But these numbers become zero if there are 10 incast senders (not shown) instead of 100. Timestamp-assisted retransmission and reiterated FIN also exhibit parameter-dependent performance. If the SRU is large, transfer time is longer and it is more likely to have double loss. Thus, we see a more significant impact on performance if we disable timestamp-assisted retransmission with SRU of 100KB. However, in this long lifetime situation, reiterated FIN packets are of less importance because the flows are more likely to converge to a fair-share bandwidth before it terminates, which makes the tail loss less frequent.

Therefore, while the combination of the three solutions make all causes of RTO to a low probability, the relative importance of each of them depends on factors including the bottleneck buffer, number of senders, and the number of packets each flow sends.

V. Discussion

A. Robustness of admission control

Extreme case of admission control: The admission control scheme in section III-A could have a possibility that all flows' handshake packets are marked, for example, when the network is severely congested. This is a rare case because, according to [25], it is unlikely for such a short-lived transfer to traverse a congested link. But when this happens, according to section III-A, all incast flows would be withheld indefinitely. Therefore, a special handling is deserved for this case.

One way to handle this situation is to withhold all flows for some random time and then gradually resumes a flow to probe for an improved network condition. Indeed, we could also impose a minimum bound on the number of concurrent flows so that at least some flows may proceed.

Multiple bottlenecks: In section III-A, it is essentially assumed that all incast flows see the same bottleneck. Therefore, when one flow finishes, it can be replaced by any withheld flow without changing the network utilization. If there are distinct bottlenecks among the flows, however, we should be careful in selecting which flow to resume as a replacement of a finished flow. As single-bottleneck is the more common case, we do not address this situation in this paper.

Accuracy of V: The admission control is guided by the prediction of future buffer usage through the virtual byte counter V and the current buffer occupancy Q. While Q shall be steady in the short period of time, V can be inaccurate. If proposals like [26] is implemented, the increment of V for the size of four packets is too small and the number of concurrent

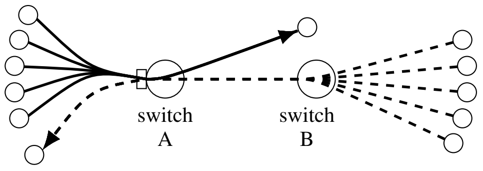

Fig. 12: Crossing incast traffic

senders will be higher than the network can support. But if the SRU is less than four packets, V would be too large and the network might be underutilized.

Therefore, we suggest the following to make V more accurate: Firstly, V shall not be increased for marked handshake packets, as they would be withheld by the incast aggregator and generate no traffic, thus they do not lead to buffer consumption in the next RTT. Secondly, there should be an upperbound for V which is comparable to threshold B. It would be unhelpful for $V > B$ as this means all new flows are denied admission. Furthermore, for a very large V, it takes a long time to 'deplete' those virtual bytes. The long time may invalidate the prediction of the network condition. We improve the accuracy by making the error of V bounded.

Background traffic: If there is a steady background traffic to share the bottleneck with incast flows, it should not have any impact to the admission control besides the effective buffer capacity is reduced. If the background traffic is a TCP flow, presumably long-lived, and thus undergoing congestion avoidance, the aggressive nature of slow start of the new incast flows will throttle it. The fairness issue between incast flows and the background TCP flow is an interesting topic.

Another incast application can also be a background traffic. Consider the case in Fig. 12, two different incast applications are sending traffic simultaneously but in different direction from the point of view of switch A. If they set up the connection at the same time, because of the nature of three-way handshake, the output port that denoted by a rectangle in the figure may observe the handshake packets of both incast traffic at any direction, but indeed, only half of those handshake packets will lead to data packets eventually while the connections of the other half only inject acknowledgment packets to that output port.

Such 'crossing' incast traffic is a rare case because their handshake processes have to start at the same time. But even if it happens, the effect would be fewer concurrent flows on each incast application. Because of the elastic nature of TCP, this means each flow could enjoy a larger share of bandwidth and completes its transfer quicker. Even each incast application takes more stages to complete all flows in each round of transfer, the total time of transfer does not lengthen a lot. Since we are not targeting for the optimal transfer time, but preventing the worse case, i.e., RTO, from happening, this phenomenon is acceptable.

B. TCP rate control in incast

TCP is not designed for incast traffic. It is known that massive synchronized TCP flows will break the statistical multiplexing: When the buffer is not yet exhausted, all flows increase their rate at the same time, which yields a large amount of packet drop at a sudden, causes every flow to have rate decrease. Moreover, the synchronized incast flows as a whole behave as one very aggressive flow. Any other flow running in parallel may not attain the fair share of bandwidth.

A way to solve this fairness issue is to do *cross-socket congestion control*, such as that of [24] or [27]. But a simple way to provide fairness is to let the incast aggregator fixes the announced receive window size to a smaller value. Since the incast aggregator keeps track on the total number of running incast flows N. We may reduce the announced receive window size from the default value of 65535 bytes to $1/N$ of it. This makes the total bandwidth used by the incast flows share the same bound as single TCP flow.

C. RED switch

Since we perform threshold-based packet marking at the switch, it is straightforward to extend the functionality of switch to perform RED [16]. We may let the switch perform RED on non-handshake packets, i.e., mark or drop packets based on queue length function, and consider the virtual byte counter only on the handshake packets.

Our preliminary study found that, RED improves the network performance. This is because the random function of RED can desynchronize flows. Then there would be less severe packet drop and thus the TCP flows may recover quicker.

VI. Previous Work

The problem of incast has been studied in several papers. It is first mentioned in [5], [28] and found to be related to the overflow of bottleneck buffer in [10]. The phenomenon is then analyzed in [29], [9], to relate it to the number of senders and the different sizes of SRU.

The earliest attempt to solve it is in the storage network setting. It is proposed in [28] to limit the number of incast senders by designing the strip policy in the storage system and throttle each sender's maximum sending rate through TCP receive window. Several similar application-level solutions are outlined by [11], which includes staggering the senders. The idea of staggering senders is further studied by [12], [13], with the close-form formula of goodput derived.

Several other solutions are outlined by [10]. These include the use of a different TCP variant for better packet loss recovery, disable slow-start to slow down the increase of TCP sending rate, or even reduce RTO_{min} to microseconds so that there would be a shorter idle time upon RTO. This last suggestion, a mitigation but not a prevention of the throughput collapse, is investigated in detail in [4], which is found to be very effective if we can make use of a high resolution timer in OS kernel. However, the availability of such timer for each TCP connection is a stringent requirement to many OS.

Throttling TCP flows to prevent the bottleneck buffer be overwhelmed is the straightforward idea. ICTCP [24] is along this line which adjusts the announced receive window according to the ingress traffic at the aggregator. However, it assumes the bottleneck is the last hop and the adjustment is based on delayed measurement data.

Throttling TCP flows can also be done in network equipments. Both [30] and [31] suggest to modify a IEEE 802.1Qau [32] switch to regulate flow rates, so that severe drop that causes incast throughput collapse is less likely to happen. In [33], two queues are used to prioritize flows with lower packet arrival rate, so that it can reduce throughput disparity and make TCP less likely to resort to RTO for recovery.

Some also tried to solve the incast problem in another direction. In [34], using a smaller MTU is suggested, which virtually increases the bottleneck buffer size in terms of number of packets. In [35], spontaneous proactive retransmission is proposed. It makes TCP senders retransmit unacknowledged packets periodically without waiting for signals from the receiver, so that timeout could be prevented, in the expense of potentially spurious retransmission and the use of an additional timer. The incast problem is understood in a different way in [7], namely, it is due to the long-lasting background traffic that shares the bottleneck with incast traffic consumes too much of the available buffer. Hence [7] proposes DC-TCP to control the background traffic so that it keeps more room in the bottleneck buffer to serve the bursty incast traffic.

VII. Conclusion

In this paper, we enumerated the three occasions that a TCP retransmission timeout may occur. Three different solutions are proposed targeting them. The three solutions, namely, admission control of TCP connections, timestamp-assisted retransmission, and reiterated FIN packets corresponds respectively to the initiation, continuation, and termination of the life of a TCP connection. We therefore refer to them as *ICaT*. They are simple, elegant solutions, as explained in section III, and easy to implement. Their objective is to assure TCP self-clocking can be maintained (i.e., each flow to keep at least a packet on the network), so that there is no need to resort to retransmission timeout to recover any lost packets.

Admission control limits the number of concurrent incast flows so that the network will not be overwhelmed. Then, we can reduce the probability of block loss as we have a smaller population to share the bottleneck. The other two solutions, timestamp-assisted retransmit and reiterated FIN packets, guarantee a packet is sent whenever an acknowledgment arrives. The former helps identifying the loss of a retransmitted packet (double loss), so that it can be retransmitted again. The latter recovers loss at the tail of a connection (tail loss), which cannot be done in a standard TCP.

We evaluated the solution and find them to be very effective in preventing the incast throughput collapse. Not only they can sustain a large number of incast senders, but also keep the transfer goodput high. Due to their simplicity in design, ICaT shall be recommended to make TCP transfer more robust in the data center environment.

References

[1] S. Shepler, M. Eisler, and D. Doveck, "Network file system (NFS) version 4 minor version 1 protocol," IETF RFC 5661, Jan. 2010.

[2] T. Haynes, "NFS version 4 minor version 2," Internet Draft, Nov. 2011.

[3] V. Naidu, "Minimum RTO values," end2end mailing list, Nov. 2004.

[4] V. Vasudevan *et al.*, "Safe and effective fine-grained TCP retransmissions for datacenter communication," in *Proc. SIGCOMM*, Aug. 2009.

[5] G. A. Gibson *et al.*, "A cost-effective, high-bandwidth storage architecture," in *Proc. 8th ASPLOS*, Oct 1998.

[6] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *Proc. 6th OSDI*, Dec. 2004.

[7] M. Alizadeh *et al.*, "Data center TCP (DCTCP)," in *Proc. SIGCOMM*, Aug. 2010.

[8] M. Allman, H. Balakrishnan, and S. Floyd, "Enhancing TCP's loss recovery using limited transmit," IETF RFC 3042, Jan. 2001.

[9] J. Zhang, F. Ren, and C. Lin, "Modeling and understanding TCP incast in data center networks," in *Proc. INFOCOM*, Apr. 2011.

[10] A. Phanishayee *et al.*, "Measurement and analysis of TCP throughput collapse in cluster-based storage systems," in *Proc. 6th FAST*, Feb. 2008.

[11] E. Krevat *et al.*, "On application-level approaches to avoiding TCP throughput collapse in cluster-based storage systems," in *Proc. Supercomputing*, Nov. 2007.

[12] M. Podlesny *et al.*, "An application-level solution for the TCP-incast problem in data center networks," in *Proc. 19th IWQoS*, Jun. 2011.

[13] H. Zheng and C. Qiao, "An effective approach to preventing TCP incast throughput collapse for data center networks," in *Proc GLOBECOM*, Dec. 2011.

[14] M. Allman, S. Floyd, and C. Partridge, "Increasing TCP's initial window," IETF RFC 3390, Oct. 2002.

[15] K. Ramakrishnan, S. Floyd, and D. Black, "The addition of explicit congestion notification (ECN) to IP," IETF RFC 3168, Sep. 2001.

[16] S. Floyd and V. Jacobson, "Random early detection gateways for congestion avoidance," *Trans. Netw.*, vol. 1, pp. 397–413, Aug. 1993.

[17] J. Postel, "Transmission control protocol," IETF RFC 793, Sep. 1981.

[18] K. Fall and S. Floyd, "Simulation-based comparisons of Tahoe, Reno, and SACK TCP," *ACM SIGCOMM CCR*, vol. 26, pp. 5–21, Jul. 1996.

[19] V. Jacobson, R. Braden, and D. Borman, "TCP extensions for high performance," IETF RFC 1323, May 1992.

[20] T. R. Henderson *et al.*, "Network simulations with the ns-3 simulator," in *Proc. SIGCOMM*, Aug. 2008.

[21] EANTC, "Cisco Catalyst 6500 with Supervisor720 — 10 gigabit ethernet performance test," http://www.cisco.com/en/US/prod/collateral/switches/ps5718/ps708/prod_white_paper0900aecd800c958a.pdf, 2003.

[22] J. Dean, "Software engineering advice from building large-scale distributed systems," Stanford CS295 class lecture, Spring 2007.

[23] Cisco, "Cisco Catalyst 4948 10 gigabit ethernet switch data sheet," Available http://www.cisco.com/en/US/prod/collateral/switches/ps5718/ps6021/ps6230/product_data_sheet0900aecd80246552.pdf, 2009.

[24] H. Wu *et al.*, "ICTCP: Incast congestion control for TCP in data center networks," in *Proc. ACM CoNEXT*, Nov. 2010.

[25] T. Benson, A. Anand, A. Akella, and M. Zhang, "Understanding data center traffic characteristics," in *Proc. WREN*, Aug. 2009.

[26] N. Dukkipati *et al.*, "An argument for increasing TCPs initial congestion window," *ACM SIGCOMM CCR*, vol. 40, pp. 27–33, Jul. 2010.

[27] P. Mehra, A. Zakhor, and C. D. Vleeschouwer, "Receiver-driven bandwidth sharing for TCP," in *Proc. INFOCOM*, 2003.

[28] D. Nagle *et al.*, "The Panasas ActiveScale storage cluster: Delivering scalable high bandwidth storage," in *Proc. Supercomputing*, 2004.

[29] Y. Chen *et al.*, "Understanding TCP incast throughput collapse in datacenter networks," in *Proc. WREN*, Aug. 2009.

[30] P. Devkota *et al.*, "Performance of quantized congestion notification in TCP incast scenarios of data centers," in *Proc. MASCOTS*, 2010.

[31] Y. Zhang and N. Ansari, "On mitigating TCP incast in data center networks," in *Proc. INFOCOM*, Apr. 2011, pp. 51–55.

[32] Data Center Bridging Task Group, "Draft Standard for Local and Metropolitan Area Networks — Virtual Bridged Local Area Networks — Amendment 7: Congestion Notification," Jul. 2009, (IEEE P802.1Qau/D2.2).

[33] A. Shpiner and I. Keslassy, "A switch-based approach to throughput collapse and starvation in data centers," in *Proc. 18th IWQoS*, 2010.

[34] P. Zhang *et al.*, "Shrinking MTU to mitigate TCP incast throughput collapse in data center networks," in *Proc. 3rd ICCMC*, 2011.

[35] S. Kulkarni and P. Agrawal, "A probabilistic approach to address TCP incast in data center networks," in *Proc. 31st ICDCS*, Jun. 2011.