TCP PLATO: Packet Labelling to Alleviate Time-Out

Shikhar Shukla, Shingau Chan, Adrian S.-W. Tam, Abhishek Gupta, Yang Xu, H. Jonathan Chao
Department of Electrical and Computer Engineering
Polytechnic Institute of New York University
Brooklyn, New York
Email: {sshukl01, schan02}@students.poly.edu, adriantam@nyu.edu, ag3987@students.poly.edu, {yangxu, chao}@poly.edu

***Abstract*—Many applications (e.g., cluster based storage and MapReduce) in modern data centers require a high fan-in, many-to-one type of data communication (known as TCP incast), which could cause severe incast congestion in switches and result in TCP goodput collapse, substantially degrading the application performance. The root cause of such a collapse is the long idle period of the Retransmission Timeout (RTO) that is triggered at one or more senders by packet losses in congested switches. In this paper we develop a packet labelling scheme PLATO, which improves the loss detection capabilities of NewReno using an innovative packet labelling system. Packets carrying this special label are preferentially enqueued, at the switch. This allows TCP to detect packet loss using three duplicate acknowledgements, instead of the time expensive RTO; thus avoiding the goodput collapse. PLATO makes minor modifications to NewReno and does not alter its congestion control mechanism. The implementation and simulations have been done in Network Simulator 3 (NS3). PLATO is found to perform orders of magnitude better than NewReno as well as state-of-art incast solution Incast Control TCP (ICTCP). We also show that TCP PLATO can be implemented using commodity switches with Weighted Random Early Detection (WRED) function.**

Keywords: TCP Incast, Data Center Networks

I. Introduction

Today's data center networks (DCNs) have become an important area of research and innovation due to the services that they provide. TCP incast goodput collapse is a critical problem that afflicts applications with high fan-in, many-to-one data transfer patterns [1]–[4], such as cluster based storage systems [5]–[7] and MapReduce [8]. The incast communication pattern was first termed by Nagle et al in [5]. In the incast scenario, a client requests a chunk of data called Server Request Unit (SRU) from several servers. These servers then send the data to the client in parallel, as shown in Fig. 1. Upon the successful receipt of all the SRUs, the transfer is completed and the client can send out new requests for another round of SRUs. Thus, the finish time of a round of transfers depends on the slowest server. For such a many-to-one data exchange in the high bandwidth, low latency DCN environment, the limited buffer space on the commodity switches becomes the critical resources [3]. Traditionally, the aforementioned applications use TCP as the transport layer protocol [2], [3]. The synchronization of various TCP flows, coupled with the bursty nature of TCP traffic and the small buffer capacity of the switches, can lead to severe packet losses, and consequently retransmission timeout

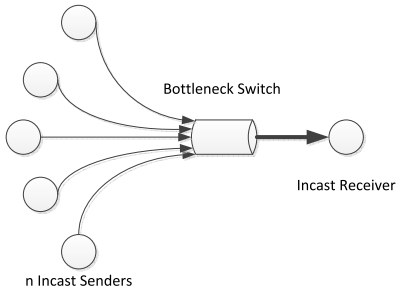

Fig. 1. Incast traffic is a high fan-in, many to one data transfer.

(RTO), at one or more TCP senders. The timeout is usually a minimum of 200 ms [9], as determined by $\mathrm{RTO}_{\min}$. In the DCN environment the typical Round Trip Time (RTT) is in the order of 100 μs, therefore the RTO can result in TCP incast goodput collapse [2].

Fig. 2 shows the link utilization characteristics for the bottleneck link. In this simulation, each of the 45 servers sends a SRU of size 100 KB each, to the client. The client and servers are using the NewReno variant of TCP [10], [11]. The switch buffer size is 256 KB and the base RTT is 100 μs. We find that initially, the link utilization is very high, with a median value of 90% of the link bandwidth. But it quickly falls to almost 0% and stays close to this value for approximately 200 ms. This corresponds to several servers experiencing a RTO in response to packet loss. After the timeout, the utilization quickly reaches the median value of 90% but falls again to almost 0%. This cycle continues throughout the life of the data transfer. Thus the RTO resulting from packet loss, leads to severe link under utilization, and consequently the goodput collapse.

Several solutions have been proposed to solve TCP incast goodput collapse problem [1], [3], [9], [12]–[14]. The solutions that have been proposed thus far, have been based on either of the two key ideologies. The first is to mitigate the adverse impact of the RTO, by reducing the default $\mathrm{RTO}_{\min}$to the 100 μs range [3]. Since $\mathrm{RTO}_{\min}$is reduced to the order of RTT, occurrence of a RTO does not cause an overall goodput collapse. The main problem with this approach is that most operating systems lack the fine grained timers required to implement the microsecond level timers. Also, every time the

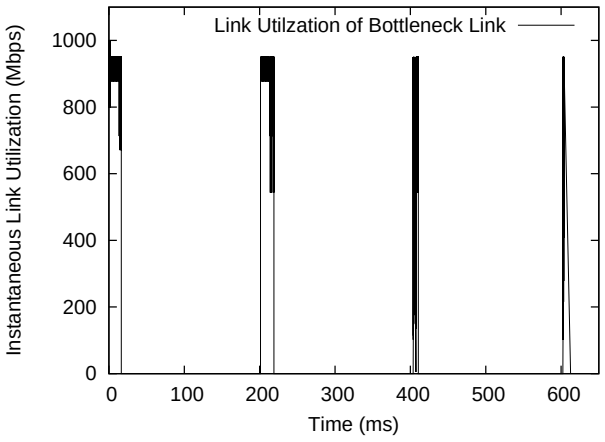

Fig. 2. Link utilization for the bottleneck link for 45 servers sending 100 KB SRU to 1 client over a 256 KB bottleneck switch buffer. All servers and client use TCP NewReno.

TCP sender experiences a RTO, it enters slow start, and its congestion window (cwnd) is reset to one full sized segment, called sender maximum segment size (SMSS). Then it takes several RTTs for the cwnd to become comparable to available bandwidth. The second ideology is to delay the build up of congestion at the switch. Approaches based on this idealogy include using ECN marks to limit sending rate of servers, receiver window based flow control and global scheduling of traffic.

In this paper, we develop a packet labelling scheme, PLATO, as a solution to TCP incast goodput collapse. We observe packet loss is not an unfavorable event for TCP[1]. It informs TCP of network congestion and allows it to regulate its data rate, as a part of the Additive Increase Multiplicative Decrease (AIMD) [15] congestion avoidance mechanism in TCP. The root cause of the incast problem lies in the inefficiency of the existing loss detection mechanisms of standard TCP NewReno, in the DCN paradigm. If TCP is able to detect loss sooner, rather than after a 200 ms timeout, the link under utilization, and thus the goodput collapse can be avoided. There exists an alternative mechanism of three duplicate acks, which allows TCP to detect loss without the 200 ms delay. But, as we show later, this method of loss detection fails under certain packet loss conditions. PLATO attempts to improve TCP's loss detection capabilities by leveraging the existing three duplicate ack mechanism in conjunction with a packet labelling system. PLATO places a special label on certain TCP segments[2]. The switch, in turn, gives a higher enqueue preference to the packets carrying these labelled segments. The TCP receiver responds to such a labelled segment with a corresponding labelled ack. The sender again sends a labelled segment on receiving a labelled ack. Even if all the unlabelled packets are dropped by the switch, the labelled segments and acks continue. This is by virtue of the preferential treatment given to these special packets, by the switch. Thus, any packet loss can be detected by means of the three duplicate acks, and the long idle period of the RTO is avoided.

PLATO makes decisions for labelling segments that will be sent in the future and relies on the continuation of the segment-ack exchange. As we show later, this depends on the cwnd, the amount of data unacknowledged in the network and the availability of data from the application layer. Owing to the dynamic network conditions, it is challenging to identify whether the segment-ack exchange can continue or not. The technical contributions of this paper are (1) Using a simple packet labelling scheme, PLATO, we are able to augment the loss detection capabilities of TCP NewReno. This allows to avoid the long idle period of RTO, and thus, the goodput collapse in incast communication is averted. (2) Without significantly modifying the behavior of TCP, PLATO is able to identify whether the segment-ack exchange can continue. (3) The simulations show that PLATO performs orders of magnitude better than NewReno and incast-specific TCP variant ICTCP. (4) We show that PLATO can be easliy implemented with commodity switches with Weighted Random Early Detection (WRED).

Section II presents the motivation for PLATO, where the relationship between acks and RTO is explained. We also enumerate the packet loss patterns that result into RTO. Section III describes PLATO in detail with the implementation described by section IV, which shows that only minor modification to the TCP stack on the end hosts is required. We also show that commodity switches with WRED functionality can be used to implement the switch functionality. Section V discusses the performance evaluation before the conclusion in section VII.

II. Motivation

TCP is a widely used transport layer protocol which provides reliable data transfer services. Although TCP has proven to be very successful in Wide Area Networks (WAN), it is unsuitable for the DCN. This is because, contrary to WAN, DCNs have a high speed, low latency network, with small buffers on switches. The small buffers facilitate the low latency as well as reduced cost. Under these network conditions, TCP's loss detection mechanism is proven inefficient and leads to severe link under-utilization, as shown in Fig. 2, and thus the goodput collapse in incast applications. PLATO attempts to improve the loss detection capabilities of TCP and make it more suitable for the DCN paradigm. Since PLATO leverages the existing mechanisms of TCP, it is prudent to analyse these in detail.

A. TCP ACK Clock

TCP uses cumulative positive acks to indicate successful receipt of segments. Acks are also used to regulate the cwnd and thus the data rate of the sender. When an ack acknowledges previously unacknowledged data, the cwnd is

[1]For the remainder of the paper, any reference to TCP should be assumed to be the NewReno variant.

[2]The unit of information sent by TCP layer to IP layer is called a segment. This segment, encapsulated in the IP header and Ethernet header and Ethernet trailer, as seen by the switch, is called a packet. For the scope of this paper, the terms *segment* and *packet*, can be used interchangeably, without loss of generality.

incremented. On the contrary, cwnd is reduced if segment loss is detected. This in accordance with the AIMD congestion avoidance mechanism. The mechanism of sending segments in response to acks is known as the TCP ack clocking. As long as there is at least one segment and/or ack in-flight (alive) in the network, the ack clock is said to be alive. If the cwnd can send out new data, and there are no segment or ack in the network, the ack clock is still said to be alive. Roughly speaking, if the ack clock is alive, the sender can send segments. This is discussed in detail in a later section.

We shall now discuss the methods used by TCP to detect loss.

B. TCP Loss Detection Mechanisms

These are heuristics adopted by TCP to recognise that a segment has been dropped by the network due to congestion. Following are the loss detection mechanisms and TCP's interpretation of the information that they provide. Note that in absence of Explicit Congestion Notification (ECN) [16] and Selective Acknowledgements (SACK) [17], it cannot be known with certainty if a particular segment was dropped due to congestion. Similarly, the level of congestion in the network cannot be accurately determined.

1) Retransmission Timeout (RTO): For every segment that is sent over a TCP connection, a countdown timer, called retransmission timer, is started. This timer is normally configured to 200ms in most Operating System (OS) implementations [9]. If an ack for the segment is not received before the corresponding timer expires, it is assumed that (i) the segment was dropped and needs to be retransmitted, and (ii) there is a *severe* network congestion. Consequently, the TCP sender drastically reduces its sending rate to one SMSS to adjust to the presumably severe congestion. The dropped segment is then immediately retransmitted.
2) Fast Retransmit (FRtx): When three duplicate acks for a segment are received before the retransmission timer expires, it is assumed that (i) the segment was dropped, and (ii) the congestion was transient, since at least three segments following the presumably dropped segment made it to the receiver. Consequently, the TCP sender halves its data rate to adjust to the presumably non-severe congestion. Also, the dropped segment is immediately retransmitted, without waiting for the 200 ms RTO.

   An important conclusion from the above discussion is that, usually, the loss of a segment will be detected by a RTO if the ack clock is dead. Otherwise, if the ack clock was alive, *most likely*, it would have resulted into the three duplicate acks required for invoking FRtx, before the expiration of the retransmission timer.

   Of the two, RTO has a much higher penalty in terms of the wasted network bandwidth owing to the 200 ms idle waiting period where the sender is unable to send any segments. This, as stated earlier, is the primary reason for the TCP incast goodput collapse. It is also noteworthy that after an RTO, the cwnd is reset to one SMSS, whereas after FRtx, the cwnd is only halved. This leads to bandwidth under-utilization, since the sender has to wait for several RTTs before the cwnd equals the available network bandwidth. Thus, three duplicate acks is the preferable loss detection mechanism, and serves as an alternative to experiencing the adverse affects of RTO. Unfortunately, the requirement of three duplicate acks for invoking FRtx cannot always be met. This is discussed in more detail in the Section II-C.

   Here, we digress briefly to discuss a very important *loss recovery* mechanism, which is intertwined with FRtx.
3) Fast Recovery (FRec): TCP enters the FRec state immediately after FRtx. FRec is an extremely robust loss recovery mechanism, whose in-built loss detection using partial acks [11], provides a very efficient way for TCP to recover from multiple segment losses in the same window.

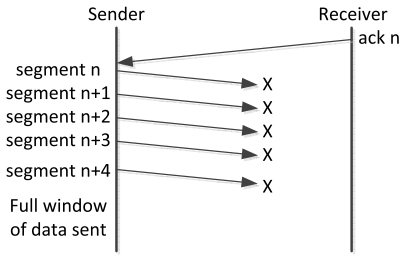

Fig. 3. Block Loss: All the segments from a window are lost.

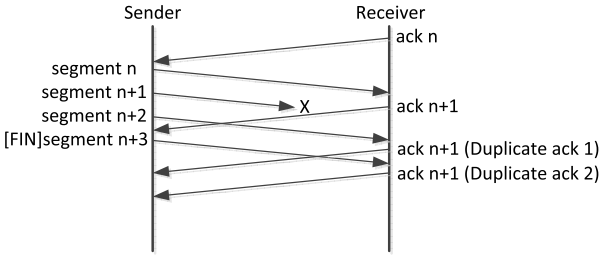

Fig. 4. Tail Loss: One of the last three segments are lost.

C. Reasons for RTO

Certain segment (and/or ack) loss patterns, may lead to the scenario, where the sender is unable to detect loss by means of three duplicate acks and suffers a RTO. These have been studied by [4] and in our previous research [18]

1) Block Loss: As shown in Fig.3 if all the segments of a window are dropped, there will be no in-flight segments or acks left in the network. This will lead to the loss of the ack clock and the cwnd will be stalled. Since there are not enough acks to invoke FRtx, the TCP sender can infer a segment loss only after the long idle period of a RTO. Note, if limited transmit [19] is not in effect; assuming that cwnd can hold at most n packets, loss of $n-2$ among any n consecutive segments can lead to such a timeout [18].

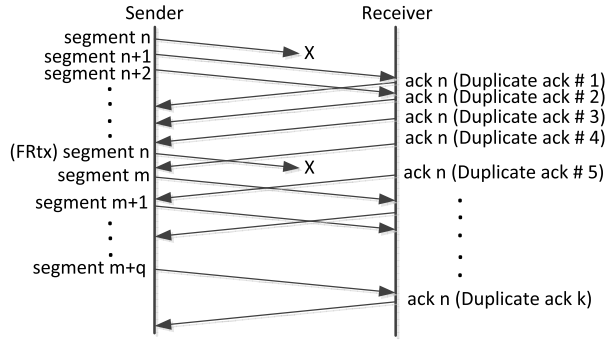

Fig. 5. Double Loss: Loss of retransmitted packet.

2) Tail Loss: As shown in Fig.4, if one (or more) of the last three segments towards the end of the life of the TCP flow are lost; the TCP receiver cannot generate enough duplicate acks to inform the sender about the loss. The ack clock eventually dies, because there are no more segments to send in response to incoming acks. Thus the TCP sender resorts to RTO to infer segment loss.
3) Double Loss: As shown in Fig.5, when the TCP sender detects a loss, the corresponding segment is retransmitted. If this retransmitted segment is dropped again, the sender will be able to detect this loss only after a RTO [10].

From the discussion above, it can be concluded that the occurrence of RTO can be attributed to two factors (i) the loss of the ack clock which renders the invocation of FRtx impossible, and (ii) the loss of retransmitted segments.

The packet labelling system adopted by PLATO, is designed to tackle both factors. As described earlier, certain packets carry the labelled segment/ack. Since the switch does not drop these special packets, the ack clock is kept alive. Thus the loss of any other packets is detected by the triple duplicate acks. A similar label allows the switch to identify the packets carrying the retransmitted packets, and preferentially enqueue them. Thus, the sender does not have to resort to the alternate, inefficient loss detection mechanism of RTO, and thus the catastrophic drop in goodput is averted. The challenge here is to identify which packet should be assigned the higher enqueue priority so that the ack-clock may stay alive. It is important to label as few packets as possible, so that the switch will have to give preference to only a small number of packets. Note that the labelling is done on the sender so the labelled packets are not an additional burden on the switch. As stated earlier, the ability to send new data depends on the cwnd, which is directly affected by the network congestion. The labelling decision is done for packets that will be sent in the future. Thus, the most challenging part of PLATO is to identify whether the sender will be able to send the packet carrying the labelled segment.

III. TCP PLATO

The primary objective of PLATO is to prevent the loss of the ack clock, so that the loss of unlabelled packets may be detected using three duplicate acks, rather than RTO. Also, to avoid RTO resulting from double loss, the packet carrying retransmitted segments must not be dropped. This goal is achieved by influencing the switch to preferentially enqueue packets carrying the labelled segment, labelled ack and retransmitted segment, while sacrificing unlabelled packets. We now present a simplistic version of the labelling scheme.

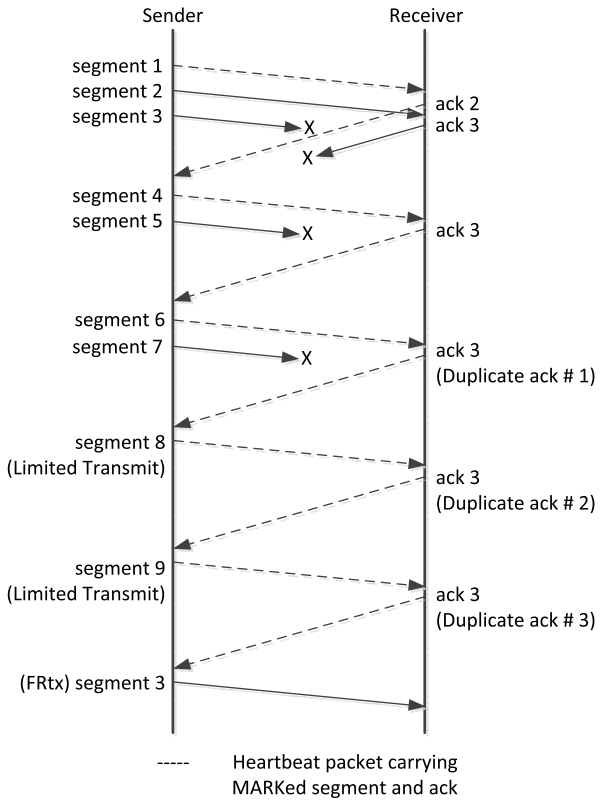

Fig. 6. A simplified implementation of PLATO. Limited Transmit [19] when enabled, allows TCP to send a segment each for the 1^{st} and 2^{nd} duplicate acks.

A. Foundation for PLATO

The ack clock is alive as long as there is at least one segment or ack in-flight in the network, or if the cwnd can send new data. Consider Fig. 6, the initial cwnd (IW) allows sending upto 3 segments. The TCP sender labels the first segment. Upon receiving this labelled segment, the receiver replies with a corresponding labelled ack. The switch, again, preferentially enqueues this packet. When the sender receives this ack, it sends another labelled segment. This process repeats, approximately once every RTT. If some (or all) unlabelled segments/acks are dropped at the switch, the labelled segment and labelled ack keep the ack clock alive. This allows the TCP sender to detect loss of the unlabelled segments using the three duplicate acks and avoid RTO. Since the packets carrying the labelled segment and labelled ack help to keep the ack clock alive, they will henceforth be referred to as the *Heartbeat Packet(s)*.

Even if it is assumed that the switch never drops heartbeat packets, the scheme described above is still based on an

implicit assumption, which is essentially flawed: *Whenever the TCP sender receives a labelled ack, it can always send a labelled segment*. It stems from the naive generalization made in Section II. This is true only if the updated cwnd allows *and* there is data to send. In the next subsection we describe the impact of receipt of acks, on the TCP NewReno cwnd, depending on the state TCP is in.

Also, consider the tail loss patterns described in Section II-C. One of the reasons a tail loss results into a RTO, is because the sender does not have data to send in response to incoming acks. Thus, even though the cwnd allows, the non availability of data becomes the limitation.

B. Response of Cwnd to Received Acks

Various acks received during Slow Start and Congestion Avoidance

1) New ack: A new ack acknowledges previously unacknowledged data. When such an ack is received during slow start or congestion avoidance, it results into an increase in the cwnd. Thus the sender is able to send previously unsent data no more than the summation of the data acknowledged by the ack and the increase in cwnd.
2) First and second duplicate acks: If limited transmit [19] is in effect, the sender is able to send one full sized segment for each of the two duplicate acks. If limited transmit is not used, the sender does not send out new data. It waits for either the third duplicate ack, a new ack or a Retransmission Timeout (RTO). For the scope of this paper, we assume that limited transmit is in effect.
3) Third duplicate ack: After the third duplicate ack is received, TCP Fast Retransmits (FRtx) the corresponding segment and enters Fast Recovery (FRec). The cwnd is halved to adjust to the network congestion. Effectively, this renders the sender unable to send any new data since the unacknowledged data is more than the updated cwnd.

Various acks received during Fast Recovery

1) Duplicate acks: These result into an artificial inflation of the cwnd by one maximum segment size each, allowing at least one segment of new data to be sent.
2) Partial acks: These acks result into partial deflation of cwnd equal to the amount of data that is acknowledged by the ack. Cwnd is also artificially inflated by one maximum segment size. The requested segment is retransmitted and new data is sent, if allowed by the updated cwnd.
3) Full ack: This pushes the sender out of FRec and into congestion avoidance. Cwnd shrinks to the value that it was at when this instance of FRec started. New data can be sent as allowed by the updated cwnd.

The above discussion can be summarized as follows.

Assume that the sender is using the NewReno variant of TCP, with limited transmit in effect, and has infinite amount of data to send. Whenever such a TCP sender receives an ack, the sender's cwnd is updated, such that it allows sending

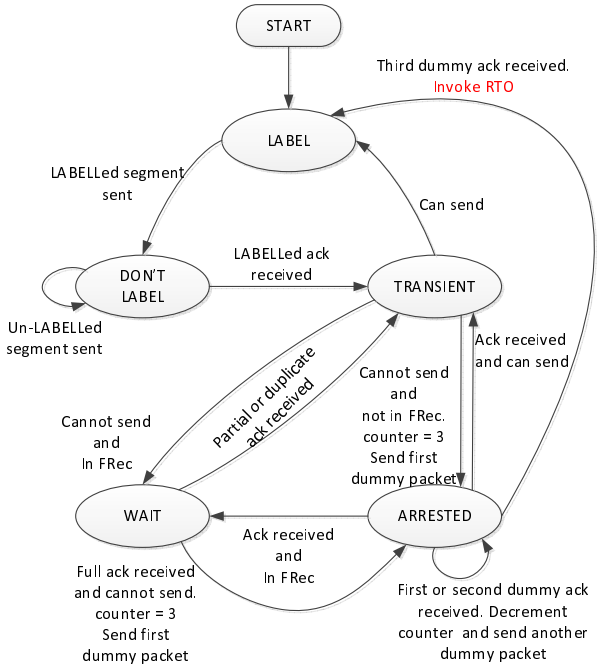

Fig. 7. PLATO state machine

out *previously unsent* data. These aforementioned acks do not include the partial ack, the third duplicate ack and the full ack. Furthermore, even if the cwnd allows, the sender will only be able to send, if data is available from the application layer.

As described in Section III-A, PLATO decides to label a segment on receiving a labelled ack. This decision is for a segment that will be sent in the future. Predicting the future cwnd is almost impossible, owing to the dynamic network conditions. Even if the current cwnd does not allow sending new data, it does not imply that the ack clock is dead. Also, the availability of application data is an important factor. PLATO is designed to be robust enough to withstand any of the possible occurrences described above. In the next subsection, we describe the complete PLATO scheme. The description is based on the scenarios that may occur and the corresponding response of PLATO.

C. PLATO State Machine

PLATO is designed as a modification to TCP NewReno and implemented as a simple state machine, shown in Fig. 7. This state machine runs in parallel with the TCP NewReno state machine. PLATO does not modify TCP's congestion control mechanism. It only decides which segments and acks should be labelled. This section describes the PLATO state machine in detail.

LABEL: After the three way handshake to establish the connection, PLATO is initialized into the LABEL state. The first data segment sent in this state is labelled. Thereafter, it promptly exits this state and enters the DON'T LABEL state.

DON'T LABEL: All the segments sent in this state will not be labelled. PLATO remains in this state until a labelled ack

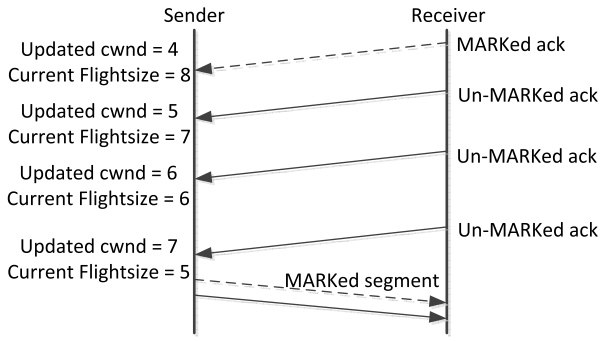

Fig. 8. Inability to send new data in response to labelled ack is not indication of stalled cwnd. Some in-flight acks may allow cwnd to send data later.

is received. This ack is generated by the TCP receiver on receiving the labelled segment, that was sent earlier. Thereafter, PLATO enters the TRANSIENT state.

The result of this interleaved labelling is that sender sends a labelled segment, once every RTT, assuming of course that there is always data to send.

The packets carrying the labelled segment/ack are the heartbeat packets. If, upon receiving a labelled ack, the TCP sender is able to send out a labelled segment, the heartbeat continues. But, as described earlier, this may not always be possible. In such a case, the heartbeat is said to be absorbed by the TCP sender, since there are no longer any labelled segments and/or acks, in-flight in the network. PLATO identifies such occurrences, in the TRANSIENT state and then takes measures to restart the heartbeat.

TRANSIENT: In this state PLATO decides whether new data can be sent, depending on the current cwnd and the availability of data from the application layer. If new data can be sent, PLATO enters label state and a labelled segment is sent. Thus the heartbeat continues. However, if no data can be sent at this moment and TCP is in FRec, PLATO either enters the WAIT state. Alternately, PLATO enters the ARRESTED state if no data can be sent at this moment, and TCP is currently in slow start or congestion avoidance.

WAIT: This state handles the case when a labelled ack is received in FRec and TCP is unable to send new data. This can happen if the received labelled ack was the third duplicate ack, a partial ack, or the sender simply has no more data to send. In such a case, PLATO *waits* for TCP to exit FRec. Suppose, during the course of this instance of FRec, receipt of an ack allows the sender to send new data, PLATO exits WAIT and enters LABEL and the heartbeat continues. However, if TCP exits FRec and the heartbeat remains absorbed, then it enters ARRESTED state. This may occur because, on receiving a full ack, the cwnd shrinks again.

ARRESTED: The ARRESTED state signifies that the heartbeat has been absorbed by the sender, while TCP is in the slow start or congestion avoidance state. Inability to send new data in response to a received ack is not an indication of a stalled cwnd or a dead ack clock. It is quite possible that some future acks, still in-flight in the network may increment the cwnd such that the heartbeat may restart itself. Such a scenario is described in Fig. 8. Thus, it is prudent to wait for data from the application layer and/or pending acks to increment the cwnd. Note that such a scenario may also arise from reordering of packets in the network. PLATO handles this as follows:

Upon entering ARRESTED state, a counter is initialized to three. A *dummy* packet is sent which carries the label but no data. On arriving at the receiver, this dummy packet is discarded and a dummy ack, carrying the label is sent back to the sender. When such a dummy ack is received and PLATO is still in ARRESTED state, the counter is decremented and another dummy packet is sent. This process is done thrice, i.e. in all, three dummy packets are sent. This procedure signifies waiting for three RTTs for any pending segments/acks on the network to arrive at either host. If the third duplicate ack is still received in ARRESTED state, it implies that the ack clock is indeed dead and the cwnd is stalled.

If standard TCP NewReno is used, without PLATO, TCP would wait 200 ms for the RTO to restart the ack clock. This is where PLATO provides a superior loss detection mechanism. Since, after the receipt of the third duplicate ack in ARRESTED state, it is clear that the ack clock is dead, the RTO can be invoked immediately. Thus PLATO is able to detect loss in at most three RTTs which are of the order of a few hundred microseconds, rather than the far more expensive 200 ms RTO.

Note that, here, the difference between TCP NewReno and PLATO is the loss detection mechanism. After the loss is detect, both PLATO and TCP would follow the same mechanism of RTO to recover from the loss.

There is good reason to implement two separate states WAIT and ARRESTED, both of which deal with absorption of heartbeat packets. FRec is a robust loss recovery mechanism. This means that if no retransmitted segment is lost, all the segments that are sent before the initiation of FRec, will be recovered in the same instance of FRec. The primary aim of PLATO is to keep the ack clock going. By ensuring that retransmitted packets are not dropped, this aim is already achieved. But slow start and congestion avoidance do not enjoy such luxury. This is why there are two separate PLATO states to handle absorption of heartbeat, depending on the current TCP state.

An important case that may arise, but has not been included in the state diagram is as follows. All the data sent has been acknowledged. A labelled ack is received, but the sender cannot send any more data since there isn't anymore data to send. In such a case, PLATO would unnecessarily send dummy packets. To handle this, a simple change is required to check whether all the previously sent data has been acknowledged. This scenario is more expected to occur in non-incast applications such as telnet.

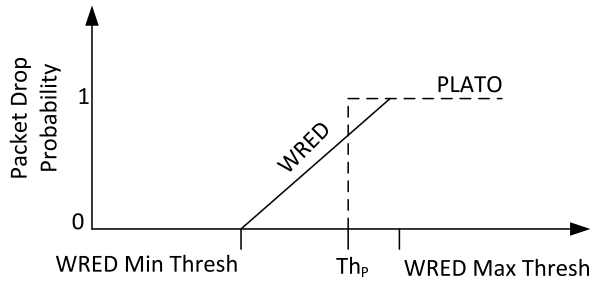

Fig. 9. WRED and PLATO drop probability for regular packets.

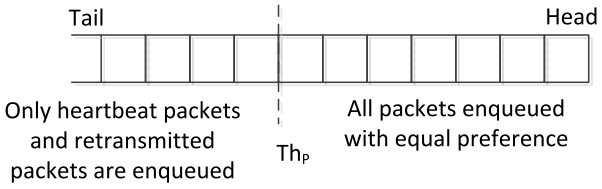

Fig. 10. Queuing mechanism for PLATO

IV. Implementation

A. Identification of special packets

The special labels that allow the switch to identify the heartbeat packets and packets carrying the retransmitted segment, are placed at the TCP sender and TCP receiver. This is achieved by using the existing Differentiated Services Code Point (DSCP) field in the IP header [20]. This field is used to prioritize traffic to achieve Quality of Service (QoS). The DSCP filed is a 8 bits long and the last two bits are used for ECN. For the switch to identify the labelled packets and retransmitted packets, only a single bit is needed in the DSCP field.

B. Switch Implementation

We had assumed in the preceding text that the switch would not drop any heartbeat packets. If standard drop tail queueing is used, eventually the buffer would overflow and the heartbeat packets may be dropped. A naive method to ensure that a heartbeat packet is never dropped would be to discard an existing unlabelled packet from a full buffer to make space for the incoming labelled packet. Although it is fine in theory, it turns out that this scheme is impractical to implement. Firstly, to find an unlabelled packet to discard from within the buffer, would require a linear search from the head of the buffer. This results into an $O(n)$ time complexity. Secondly, to discard a packet, would require the maintenance of a double linked list, instead of the FIFO, for the packets. This significantly increases the number of memory access required for storing any packet in the buffer.

We present a modified buffer management scheme, shown in Fig. 10, to support PLATO. It employs a simple FIFO to enqueue and dequeue the packets. There exists an intermediate threshold Th_P. If the instantaneous queue length is less than Th_P, all the incoming packets are enqueued with equal preference. On the other hand, if the instantaneous queue length is greater than or equal to Th_P, only the packets carrying the labelled segment/ack, and retransmitted segment will be enqueued. Thus the switch *preferentially enqueues* certain packets beyond Th_P. This simple modification allows the switch to differentiate between the incoming packets and allow the senders to avoid RTO.

Cisco IOS 12.0 supports a queuing mechanism called Weighted Random Early Detection (WRED). This uses the bits of the DSCP field to establish IP precedence. A packet with higer IP precedence is given a greater enqueue preference. The WRED feature is already available on low end cisco catalyst 3550 series switches which cost less than $500. This feature can be easily configured for PLATO.

PLATO uses the instantaneous queue occupancy rather than the average queue occupancy used by WRED. (1) is the exponential moving average used by WRED to calculate the average queue length l.

$$l_{\text{new avg}} = (l_{\text{old avg}} \times (1 - n)) + (l_{\text{instantaneous}} \times n) \quad (1)$$

For n = 1, the average will be equal to the instantaneous queue length.

As shown in Fig. 9, unlike WRED, PLATO does not use a probability to decide whether a packet should be dropped or not. Such functionality can be easily achieved by making the minimum threshold and maximum threshold of WRED, equal to the PLATO threshold Th_P.

V. Performance Evaluation

In this section we study the impact of PLATO on the performance of TCP NewReno in the incast scenario. We compare PLATO's performance with state of art incast solutions Incast Control TCP (ICTCP) [13] and Data Center TCP (DCTCP) [12]. We also compare PLATO's performance against standard Tcp NewReno with reduced values of $\text{RTO}_{\min}$, 10 ms and 1 ms. All the simulations have been performed in a NS3 testbed with the star topology of Fig.1. The link bandwidth is 1 Gbps and the link delay is 25 μs making the base RTT 100 ms. In all the simulations, the application at the incast senders sends 1 SRU to the incast client. The application sends the data as quickly as possible to the TCP layer which is then responsible for ensuring successful delivery. The client application considers the job as complete only when all the senders have successfully transferred all their data to the client. The switch, which is shared by all the flows, employs the WRED functionality mentioned earlier with the minimum and maximum threshold set to Th_P. Each simulation is run for 100 rounds and the data represents the 95^{th} percentile. A different random seed is used for each round. The metric for performance comparison is the goodput, which is calculated as the ratio of the total data transferred by all the servers to the client and the time required to complete the data transfer.

Fig.11 and Fig.12 show the performance of PLATO for short lived flows and long lived flows respectively. The simulations have been done for Th_P values of 0.3, 0.5 and 0.7 of maximum queue length. We observe that in both long lived flows and

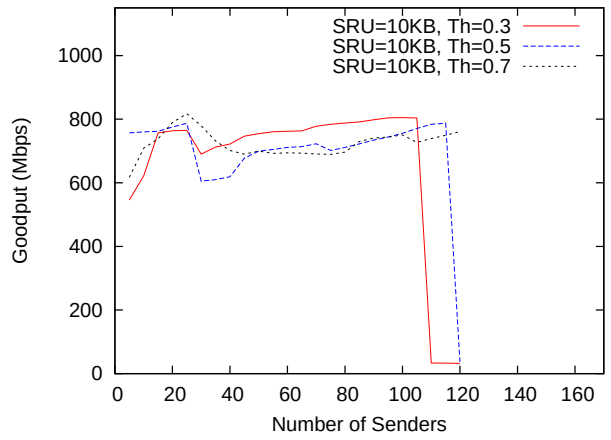

Fig. 11. Effect of Th_P on the performance of PLATO for short lived flows. Switch buffer is 256 KB and SRU is 10 KB

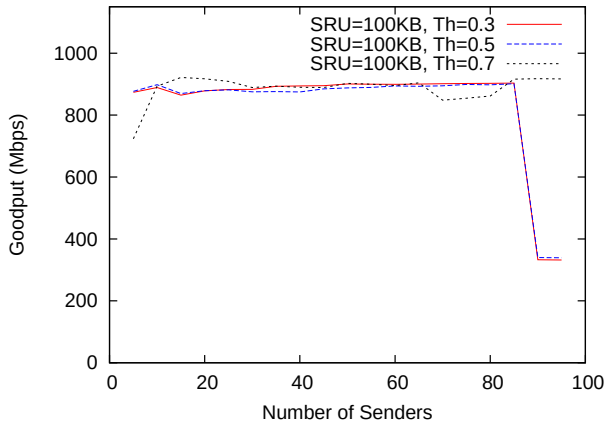

Fig. 12. Effect of Th_P on the performance of PLATO for long lived flows. Switch buffer is 256 KB and SRU is 100 KB

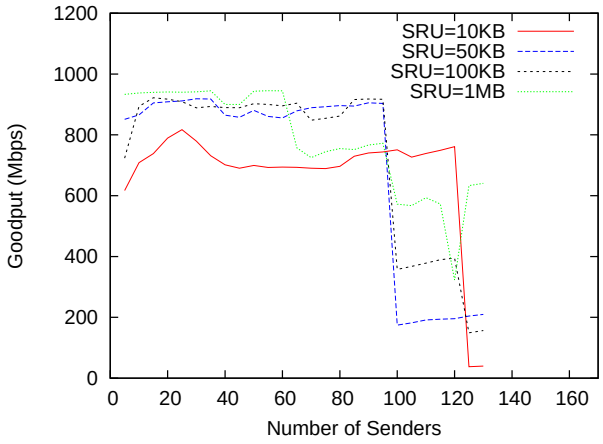

Fig. 13. Performance of PLATO for switch buffer size of 256 KB, under different SRU sizes. Th_P is 0.7

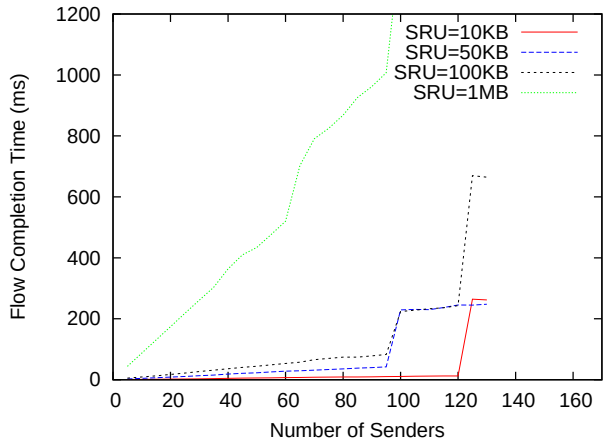

Fig. 14. Flow completion time, for PLATO flows. Switch buffer size is 256 KB and Th_P is 0.7

short lived flows, the value of the Th_P does not have a significant impact on the overall performance. PLATO is easily able to support over 90 senders, for long lived flows, before a significant loss in goodput. For short lived flows, this number is even larger, around 120 senders. Also, PLATO is able to achieve nearly 90% link bandwidth for long lived flows. We observe that the catastrophic drop in throughput for the PLATO simulations is owing to the loss of heartbeat packets. Since the switch buffer size is limited, eventually, the space reserved for the heartbeat packets, i.e. beyond the Th_P, becomes full. This leads to drop of heartbeat packets which causes PLATO to rollback to TCP NewReno, since there are no more heartbeat packets in the system.

Fig.13 shows the performance of PLATO for switch buffer size of 256 KB and Th_P 0.7, for various SRU sizes. We find that for SRU sizes 50 KB and 100 KB, the performance is nearly identical. The performance for SRU 10 KB is slightly lower due to link under utilization. For SRU size 1 MB, the performance is identical to SRU 100 KB and 50 KB till about 60 senders. Thereafter, we see a sudden drop in goodput. This is because of one or more senders experiencing a 200 ms timeout. For 1 MB SRU, owing to the large SRU size, the flow completion time is of the order of 100 ms, even if a sender doesn't experience a timeout. Thus the goodput does not drop to very low values despite the 200 ms timeout. For 10 KB SRU and 100 KB SRU, the flow completion time is of the order of 10 ms. Thus the effect of a timeout event is more prominent in 100 KB SRU and 50 KB SRU curves, as is evident by the flow completion times shown in Fig.14.

Fig.15 shows the performance of PLATO under conditions of small switch buffer size. The buffer size for this simulation is 64 KB and Th_P is 0.7. We find that the for SRU 10 KB and 100 KB, the number of senders that PLATO can support without incast is much lower in a smaller switch buffer (64 KB) than a larger switch buffer (256 KB). This is because, when the buffer size is small, it quickly overflows, leading to packet loss. Unless heartbeat packets are dropped, PLATO detects the packet loss at the switch and retransmits the corresponding packets. Since Th_P value is 0.7, switch buffer space equivalent to (64 * 0.3) KB is reserved for heartbeat and retransmitted packets. Any sender can have at most 1 heartbeat packet and 1 retransmitted packet in the network. Assuming

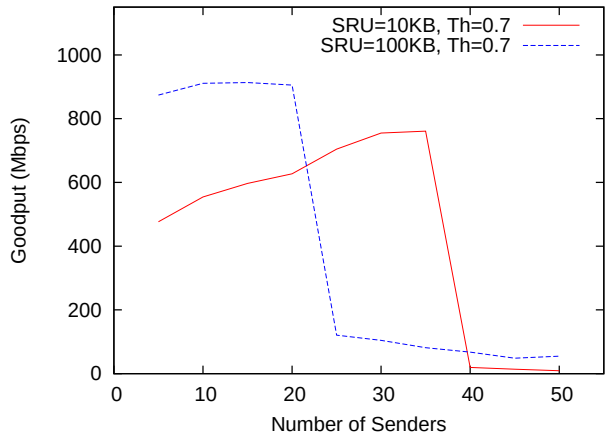

Fig. 15. Performance of PLATO for switch buffer size of 64 KB. Th_P is 0.7

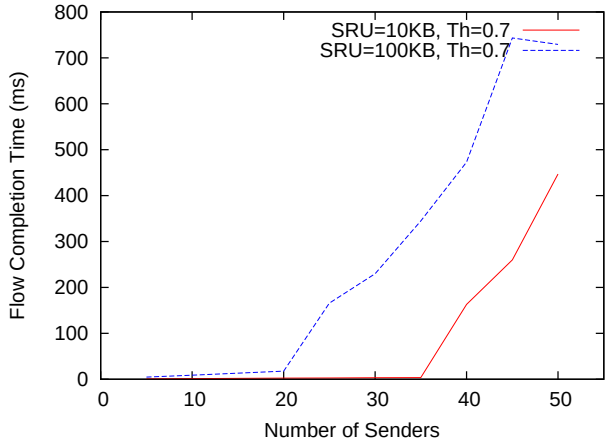

Fig. 16. Flow completion time, for PLATO flows. Switch buffer size is 64 KB and Th_P is 0.7

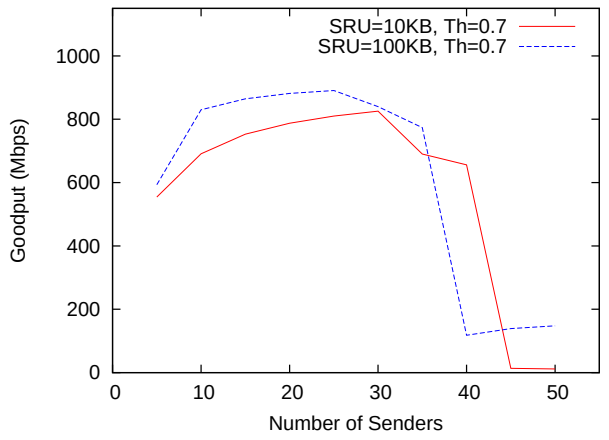

Fig. 17. Performance of PLATO with background TCP NewReno traffic. Switch buffer size is 256 KB. Th_P is 0.7

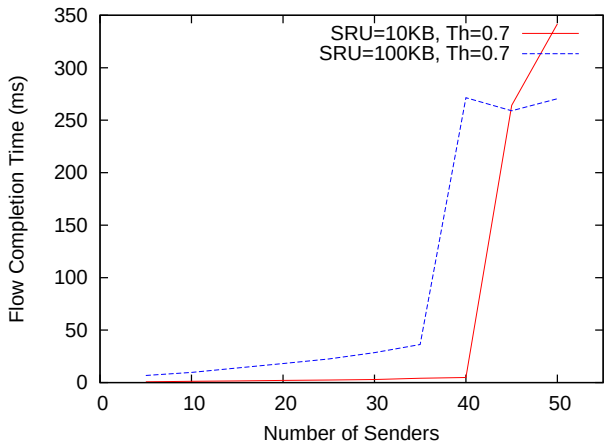

Fig. 18. Flow completion time, for PLATO flows, when background TCP NewReno flows are present. Switch buffer size is 256 KB and Th_P is 0.7

that both these packets are of maximum size 1500 bytes, at least 6 senders that can be supported by this switch in the worst case. Our simulation results show that in fact the number of senders that are supported are much larger, around 20 for 100 KB SRU and 35 for 10 KB SRU. Thus the limiting factor is the space reserved for the heartbeat packets. Fig.16 shows the corresponding flow completion time for PLATO flows. We observe that the flow completion time for the 100 KB SRU curve increases dramatically after 20 senders. Since the buffer space is very small, the hearbeat packets get dropped several times during the life of an individual flow leading to successive 200 ms timeouts on a sender, and thus a large flow completion time.

Fig.17 shows the performance of PLATO in the presence of background flows. In this simulation, several PLATO incast flows share the bottleneck link with 5 background TCP NewReno flows. The application at the TCP NewReno senders is the same as for PLATO senders. We find that PLATO flows are able to achieve a high goodput, but the number of flows that can be supported without incast is reduced. This is because PLATO relies on availability of switch buffer space. In the presence of background traffic, the switch buffer available is reduced an thus the number of incast senders that can be supported is also reduced. Fig.18 shows the corresponding flow completion time for PLATO flows.

Fig.19 and Fig.20 show the performance of PLATO compared to ICTCP, DCTCP and TCP NewReno with reduced $\text{RTO}_{\min}$ values of 10 ms and 1 ms. The switch buffer is 256 KB with Th_P value 0.7. Sender SRU size is 10 KB for simulations of Fig.19 and 100 KB for Fig.20. We find that while the performance of ICTCP and DCTCP degrade very quickly as the number of senders increases, PLATO is able to support a much larger number of incast senders, without throughput collapse. In Fig.19, PLATO is easily able to achieve a goodput of 70% of link bandwidth, while it is even higher, around 90% in Fig.20, as the value of SRU is higher. We observe that with such improved performance of PLATO is attributed to the improved loss detection capabilities.

We also note that the apparent poor performance of ICTCP and DCTCP in our simulations is attributed to the difference in

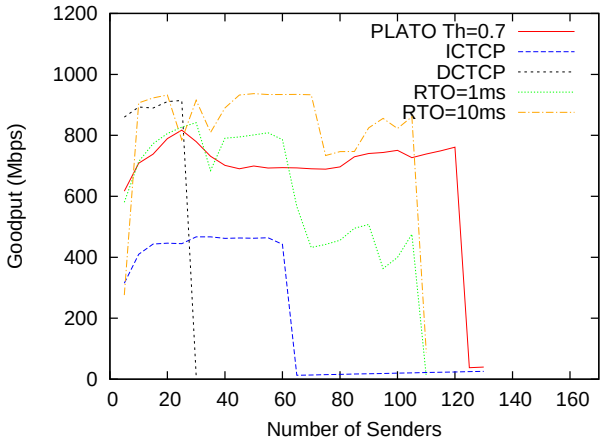

Fig. 19. Performance of PLATO compared to DCTCP, ICTCP and TCP NewReno with reduced RTO_{min} values of 10 ms and 1 ms. Switch buffer is 256 KB and SRU is 10 KB. Th_P is 0.7

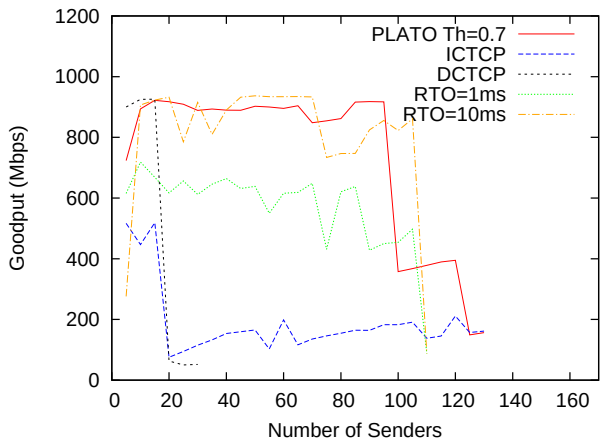

Fig. 20. Performance PLATO compared to DCTCP, ICTCP and TCP NewReno with reduced RTO_{min} values of 10 ms and 1 ms. Switch buffer is 256 KB and SRU is 100 KB. Th_P is 0.7

switch buffer size, compared to that used in [13][3] and [12][4], respectively. This is supported by the simulations of Fig.21 where a switch buffer of 2 MB is used. Here, we observe that performance of both DCTCP and ICTCP improve compared to simulations with smaller buffer size of 256 KB. Still, performance of PLATO is significantly better, as it achieves nearly 90% of the link bandwidth. The number of incast senders that can be supported is also much higher.

In Fig.19, Fig.20 and Fig.21, we observe that the performance of Tcp NewReno with reduced RTO_{min} value of 10 ms is consistently comparable to that of PLATO. This is expected since the reduced RTO_{min} value essentially reduces the idle time for the sender. Thus, even though several senders may experience timeout, the loss in goodput is not high. But it is worth mentioning that the reduced RTO_{min} value requires the availability of a fine grained timer. PLATO is able to achieve the same effect using heartbeat packets.

Fig.22 shows the instantaneous queue occupancy characteristics for the incast setting of 45 senders sharing a 128 KB bottleneck switch buffer. Fig.23 shows the queue occupancy for the incast setting of 80 senders sharing a 256 KB bottleneck switch buffer. We see that at the very beginning, almost all the packets in the switch are the heartbeat packets. This corresponds to the senders sending the first LABELLed segment. The dip in the queue occupancy corresponds to the first instance of congestion where several senders suffer packet loss. But the senders recover quickly using the three duplicate acks. Thereafter, the queue occupancy stays close to the threshold value. This shows that there are several packets dropped. But the senders don't suffer a RTO.

We also observe that the heartbeat packets are nearly 50% of the total load on the switch. But it is important to note that the heartbeat packets are not an additional load on the switch. In fact, the heartbeat packets allow the switch to prioritize

[3]ICTCP simulations in [13] use a Quanta LB4G switch.

[4]DCTCP simulations in [12] use switches with buffer sizes of 4 MB and 16 MB

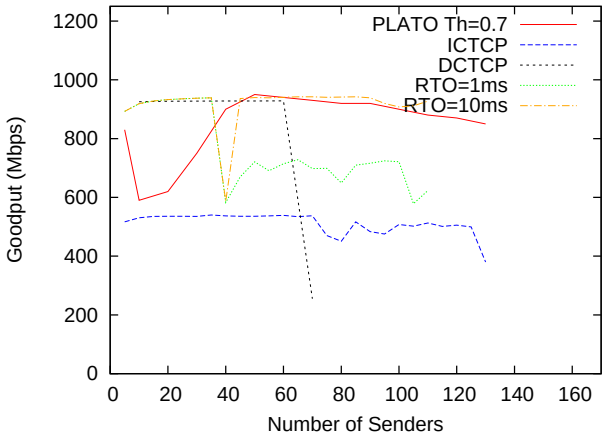

Fig. 21. Performance of PLATO compared to DCTCP, ICTCP and TCP NewReno with reduced RTO_{min} values of 10 ms and 1 ms. Switch buffer is 2 MB and SRU is 100 KB. Th_P is 0.7

packets from amongst its existing load, so that the senders can avoid RTO.

Fig.24 shows the instantaneous cwnd for an incast sender with PLATO in effect. We observe that on several occasions, the cwnd is reduced to half its original value. This corresponds to loss detection and invocation of FRec. But the sender doesn't suffer from RTO.

VI. Related Work

TCP incast throughput collapse is a widely studied and well understood phenomenon. Incast was originally observed and coined in [5]. The problem has been studied in [1], [2], [4] to analyze the relationship between throughput collapse and factors like the environment, topology factors, switch buffer size, number of incast servers and size of Server Request Unit (SRU).

Several papers attempted to solve the incast problem. For example, [5] suggests to limit the number of incast senders so as to prevent the queue buildup, and [21] gives a list of application level strategies to work around the network

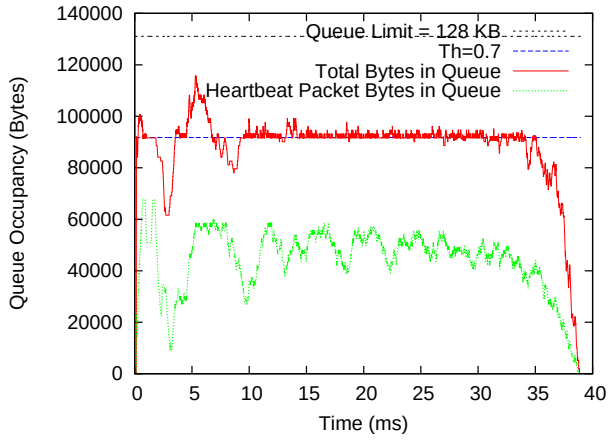

Fig. 22. Instantaneous queue occupancy for incast setting of 45 senders sharing a 128 KB bottleneck switch buffer. SRU is 100 KB

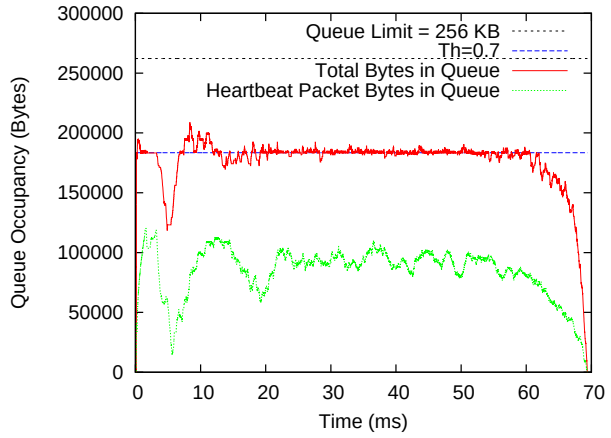

Fig. 23. Instantaneous queue occupancy for incast setting of 80 senders sharing a 256 KB bottleneck switch buffer. SRU is 100 KB

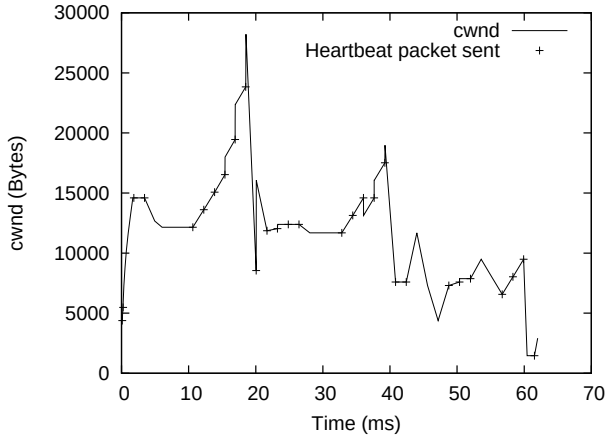

Fig. 24. Instantaneous cwnd for an incast sender. Time instants when a heartbeat packet is sent is indicated

limitation. Different solutions are explored by [2], including the effectiveness using larger switch buffers, increasing SRU, using other TCP variants, reducing the duplicated ack threshold for triggering fast retransmit, or reducing the penalty of retransmission timeouts. But the authors found none of these can solve the throughput collapse convincingly whereas each of them suffer from implementation difficulties. The idea of using an RTO of orders of magnitude smaller, to make it comparable to round trip time (RTT), is studied in detail by [3]. It is straightforward, but has a major drawback of requiring a fine-grained timer to implement it, which is not guaranteed by the operating system or at a high computation cost.

Some proposals target the protocol. In [22], a modified TCP retransmits unacknowledged packets probabilistically without confirming their loss by three duplicated acknowledgement. This tackles the problem of losing acks or massive loss of segments in a short period of time, but it also requires a timer to send packets. Thus it also needs a fine-grained timer to be useful in a short RTT environment.

ICTCP [13] tries to regulate the total sending rate of the incast transfer to prevent overwhelming the bottleneck. It adjusts the receive window size of each connection to make sure the aggregated rate is within bound. However, implementing this requires an additional layer on top of TCP layer to find the aggregate rate and adjust them accordingly. Moreover, this is a incast-specific protocol which means an incast application needs to explicitly choose to use this TCP variant. Moreover ICTCP has the limitation that it can only handle the last hop congestion.

DCTCP [12] assumes the incast throughput collapse is due to the coexistence of incast traffic with long-lasting background traffic, such that the latter consumes too much bottleneck buffer. Therefore, it tries to gather data from the Explicit Congestion Notification (ECN) bit of packets over a round trip time to get a better sense of the network congestion and react accordingly. However, as we can see in our experiments, incast throughput collapse exists even if it is not competing with background traffic for network bandwidth.

VII. Conclusion

TCP incast goodput collapse is caused by retransmission timeout (RTO) at one or more senders, in response to dropping of packets at the switch, in datacenter networks (DCN). The dropping of packets is due to several parallel TCP flows overflowing the limited buffer space on the commodity switches used in DCN. We propose a packet labelling scheme called PLATO, which improves the loss detection capabilities of TCP NewReno in the DCN paradigm. PLATO, when used in conjugation with Weighted Random Early Detection (WRED) functionality at the switch, allows TCP to detect packet loss using three duplicate acks, instead of the expensive RTO. The simulation results show that performance of PLATO is orders of magnitude better than state-of-art incast solution Incast Control TCP (ICTCP).

REFERENCES

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

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

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

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

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

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

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

[8] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, Jan. 2008.

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

[10] V. P. M. Allman and E. Blanton, "TCP congestion control," IETF RFC 5681, Sep. 2009.

[11] A. G. T. Henderson, S. Floyd and Y. Nishida, "The newreno modification to TCP's fast recovery algorithm," IETF RFC 6582, Apr. 2012.

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

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

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

[15] D.-M. Chiu and R. Jain, "Analysis of the increase and decrease algorithms for congestion avoidance in computer networks," *Comput. Netw. ISDN Syst.*, vol. 17, no. 1, pp. 1–14, Jun. 1989. [Online]. Available: http://dx.doi.org/10.1016/0169-7552(89)90019-6

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

[17] S. F. M. Mathis, J. Madhavi and A. Romanow, "Tcp selective acknowledgment options," IETF RFC 2018, Oct. 1996.

[18] A. S.-W. Tam, K. Xi, Y. Xu, and H. J. Chao, "Preventing tcp incast throughput collapse at the initiation, continuation, and termination," in *Proceedings of the 2012 IEEE 20th International Workshop on Quality of Service*, ser. IWQoS '12. Piscataway, NJ, USA: IEEE Press, 2012.

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

[20] F. B. K. Nichols, S. Blake and D. Black, "Definition of the differentiated services field (ds field) in the ipv4 and ipv6 headers," IETF RFC 2474, Dec. 1998.

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

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