Understanding TCP Incast Throughput Collapse in Datacenter Networks

Yanpei Chen, Rean Griffith, Junda Liu, Randy H. Katz
RAD Lab, EECS Dept. UC Berkeley
{ychen2,rean,liujd,randy}@eecs.berkeley.edu

Anthony D. Joseph
Intel Labs Berkeley
anthony.joseph@intel.com

ABSTRACT

TCP Throughput Collapse, also known as Incast, is a pathological behavior of TCP that results in gross under-utilization of link capacity in certain many-to-one communication patterns. This phenomenon has been observed by others in distributed storage, MapReduce and web-search workloads. In this paper we focus on understanding the dynamics of Incast. We use empirical data to reason about the dynamic system of simultaneously communicating TCP entities. We propose an analytical model to account for the observed Incast symptoms, identify contributory factors, and explore the efficacy of solutions proposed by us and by others.

Categories and Subject Descriptors

C.2.5 [**Computer-communication Networks**]: Local and Wide-Area Networks—*Internet*

General Terms

Networks

Keywords

TCP, Throughput Collapse, Incast, Unix

1. INTRODUCTION

Internet datacenters support a myriad of services and applications. Companies like Google, Microsoft, Yahoo, and Amazon use datacenters for web search, storage, e-commerce, and large-scale general computations. Business cost efficiencies mean that datacenters use existing technology. In particular, the vast majority of datacenters use TCP for communication between nodes. TCP is a mature technology that has survived the test of time and meets the communication needs of most applications. However, the unique workloads, scale, and environment of the Internet datacenter violate the WAN assumptions on which TCP was originally designed. For example, in contemporary operating systems such as Linux, the default RTO timer value is set to 200ms, a reasonable value for WAN, but 2-3 orders of magnitude greater than the average roundtrip time in the datacenter. As a result, we discover new shortcomings in technologies like TCP in the high-bandwidth, low-latency datacenter environment.

One communication pattern, termed "Incast" by other researchers, elicits a pathological response from popular implementations of TCP. In the Incast communication pattern, a receiver issues data requests to multiple senders. The senders, upon receiving the request, concurrently transmit a large amount of data to the receiver. The data from all senders traverses a bottleneck link in a many-to-one fashion. As the number of concurrent senders increases, the perceived *application-level throughput* at the receiver collapses. The receiver application sees *goodput* that is orders of magnitude lower than the link capacity.

The incast pattern potentially arises in many typical datacenter applications. For example, in *cluster storage* [8, 10], when storage nodes respond to requests for data, in *web-search*, when many workers respond near simultaneously to search queries, and in batch processing jobs like *MapReduce* [7], in which intermediate key-value pairs from many Mappers are transferred to appropriate Reducers during the "shuffle" stage.

In our opinion, a thorough solution to address the Incast pathology would have several elements: 1) a demonstration that the problem is general, and not limited to particular network environments; 2) an analytical model that identifies the likely causes of the problem and predicts the experimentally observed Incast symptoms; 3) modifications to TCP that help mitigate the problem, implemented in OS kernels and evaluated using real life workloads. Ideally, a thorough solution would also provide signatures or benchmarks that help diagnose systems for the presence of Incast.

In this paper we make progress towards a thorough solution. Our contributions are: we reproduce the results in prior work in our own experimental testbeds and offer another demonstration of the generality of Incast; we propose a quantitative model that accounts for some of the observed Incast behavior and provide qualitative refinements that give plausible explanations for the other symptoms we observed; we implement several minor, intuitive modifications to the TCP stack in Linux, and demonstrated that some modifications are more helpful than others. Based on our analytical model and experimental results, we comment on other potential TCP solutions, as well as the likely impact of mitigation techniques in application and network layers.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
WREN'09, August 21, 2009, Barcelona, Spain.
Copyright 2009 ACM 978-1-60558-443-0/09/08 ...$10.00.

The rest of the paper is organized as follows. Section 2 presents background on the Incast phenomena, Section 3 describes the workloads and testbeds we used. Sections 4, 5 and 6 present a detailed discussion of the dynamics of incast, with a simple mathematical model to explain our empirical observations and possible strategies for eliminating Incast. Section 7 summarizes our findings and outlines future work.

2. BACKGROUND

The direct predecessor to our work looked at TCP Incast in cluster storage systems [10]. The Incast pathology was observed in a variety of settings. The authors evaluated a spectrum of TCP variants, including TCP Reno, New Reno, and SACK. None of them helped. Eliminating TCP slow start did not help either. Reducing the minimum value of the retransmission timeout (RTO) timer from the default 200ms to 200μs significantly alleviates the problem in simulations. However, as the authors pointed out, most systems lack the high-resolution timers required for such low RTO values. We will show later that reducing the minimum RTO timer is nevertheless very promising. The authors also attempted a variety of non-TCP work-arounds and identified the shortcomings of each. Increasing the size of switch and router buffers delays the onset problem to configurations with more concurrent senders. However, switches and routers with large buffers are expensive, and even large buffers may be filled up quickly with ever higher speed links. Ethernet flow control removes the problem for a single switch topology, but creates head-of-line blocking and breaks down in multi-switch or multi-layered switching topologies. Application level solutions are possible, e.g. global scheduling of requests, but requires potentially complex modifications to many key applications that use TCP.

More recent work from the same authors proposed solutions centered on fine grained OS timers to facilitate sub-millisecond RTO timer values, randomizing RTO timers, and disabling TCP delayed ACKs [11]. The workload they used changed significantly from the workload used in their earlier work in [10]. In Section 3.1 we discuss the impact of different workloads. Some of our findings here depart from the findings in [11], likely caused by the difference in workloads. We have been in regular discussions with the authors, and exchanging workloads for evaluation on each other's testbeds is an ongoing research effort.

A key shortcoming of the prior work is the lack of an analytical model to understand the Incast phenomenon.

Since we are altering TCP mechanisms, it would be instructive to trace back the watersheds in our understanding of TCP. Work in [6] proved that the additive increase, multiplicative decrease algorithm used in all TCP implementations would eventually converge to an efficient and fair state. The work was considered a theoretical watershed. The original TCP congestion avoidance and control algorithms used intuitions from linear system control theory to motivate the exponential backoff mechanism [9]. Original implementations of the control algorithm used ad-hoc parameter values, usually multiples of two, to facilitate faster OS kernel code using bit-shift multipliers. These insights guided us when we designed and implemented our TCP modifications.

Driven by cost concerns, other research on Internet datacenters has seen increased focus on commodity and readily available technology [3]. Cost concerns and the preference for existing technology mean that solutions like Infiniband and custom transport protocols are less attractive. Hence our focus on TCP solutions.

3. METHODOLOGY

3.1 Workload

We use a workload inspired by distributed storage applications and bulk block transfers in batch processing tasks such as MapReduce. The workload is as follows. The receiver requests k blocks of data from a set of S storage servers – in our experiments $k = 100$ and S varies from 1 to 48. Each block is striped across S storage servers. For each block request received, a server responds with a fixed amount of data (fixed-fragment workload). Clients do not request block $k + 1$ until all the fragments of block k have been received – this leads to a *synchronized read pattern* of data requests. We configure each storage server to use block fragments of size 256KB, i.e. each block requested equals ($256KB * S$ senders) bytes. This workload is identical to the workload in [10], and we re-used their code there.

In our experiments, the metric of merit is *application-level throughput* (**goodput**), given by the total bytes received from all senders divided by the finishing time of the *last* sender.

In their latest work, the authors of [10] switched to a different workload [11]. In the new workload, the block size is fixed, instead of the fragment size being fixed (variable-fragment workload). As the number of senders increase, each sender would transmit an ever decreasing fragment of data. The new workload was thought to be more representative of communication patterns in popular distributed storage systems. However, we believe that the original workload is more representative of communication patterns in other applications involving bulk data transfer. Also, using the variable-fragment workload, as we increase the number of servers, we would eventually encounter a situations in which each server sends only one or two packets of data. Consequently, any Incast behavior could be masked by the small data transfer size for each server.

3.2 Testbed and Tools

We conducted our experiments on two platforms. First, we used a small scale local cluster with 2.2GHz, quad-core, 64-bit x86 machines running Linux 2.6.18.8. The network is 1Gbps, and all servers are connected through single-layer switching, using a Nortel 5500 series switch. To the best of our knowledge, the Nortel 5500 series switch has a maximum of 131072B of buffer space for each port, and 786432B total buffer space shared between 48 ports.

We conducted the bulk of our experiments on the DETER Lab testbed [5]. Here, we have full control over the nodes, the network topology and the connection speed between nodes. This flexibility allows us to run custom operating system images, configure network bandwidth, and influence the network hardware used. Unless otherwise noted, we used 3GHz dual-core Intel Xeon machines with 1Gbps network connections, connected using single-layer switching through a Nortel 5500 series switch. These settings ensured comparability with our results from the small scale local cluster.

For data collection and analysis we used a combination of tools. We used tcpdump [1] and tcptrace [2] to collect and analyze TCP data. We also built our own a timeline

reconstruction and analysis tool in Java, to fill up gaps in the analytical capabilities of tcptrace. These tools are not yet sufficiently polished to be released as a contribution to the general research community.

3.3 Exploratory Approach

Despite the existence of prior work on Incast, we felt that we were still at an early stage in understanding the problem. We were hesitant to prescribe a rigid list of controlled and varied parameters, and we felt that a potential contribution would be to narrow down the parameter space relevant to Incast. Therefore, our priority was to understand the Incast problem, instead of implementing and evaluating the widest range of possible solutions. We let our findings at each step guide our explorations in the next step. We also avoided simulations, because we believe that event driven simulations like ns-2 may inadequately reflect the timing ambiguities and non-deterministic variations that could affect simultaneous bulk data transfers.

4. INITIAL FINDINGS

The first step in our study of Incast is to replicate the problem in our test environment to convince ourselves that the Incast phenomenon is general. Our measurements are in Figure 1, overlaid on top of the findings in [10]. The curves are nearly identical.

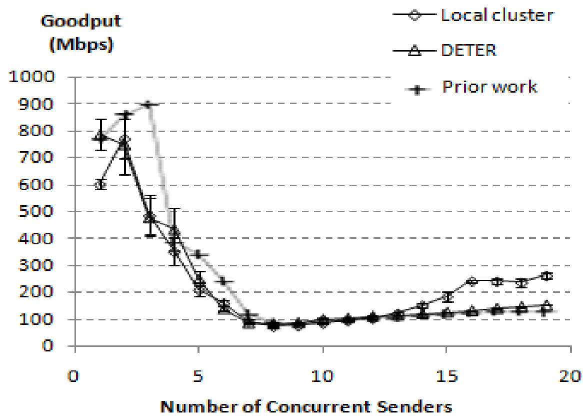

Figure 1: TCP Incast goodput collapse up to 20 senders for three different environments

A cursory look at the sequence number graphs shows that different senders experience long, synchronized TCP retransmission timeout (RTO) events. The sequence number graphs from two typical senders are shown in Figure 2 reproduce from our earlier work. Each RTO event lasts for 200ms, the default minimum RTO timer value for the Linux TCP stack. The 200ms default value was originally designed for WAN environments. It is significantly larger than average round trip time in the datacenter, typically a sub-millisecond value.

These observations inspired us to attempt a series of minor, intuitive modifications to the Linux kernel. The modifications included decreasing the minimum TCP RTO timer from the default 200ms, randomizing the minimum TCP RTO timer value, setting a smaller multiplier for the RTO exponential back off, and using a randomized multiplier for

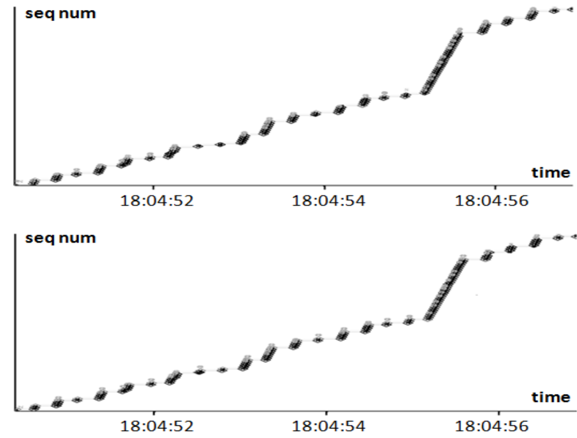

Figure 2: TCP sequence numbers vs. time for two senders in a 5-to-1 setup

the RTO exponential back off. There was another intuitive modification that we did not attempt - randomize each the timer value of each RTO as they occur. We believe this modification was redundant, because Linux TCP initially sets the TCP RTO timer to its minimum value. We believe that if the initial RTO timer is randomized, then the timer values for all subsequent RTO events would be randomized. All the modifications required replacing one line of code in the Linux kernel, sometimes with several lines of new code[1]. We randomize various timer values by using existing randomness in TCP write sequence numbers, which was originally implemented to prevent sequence number guessing attacks [4].

We implemented these modifications and ran the fixed-fragment and variable-fragment block transfer workloads in modified Linux kernels on DETER. We quickly discovered that many of these modifications were unhelpful. The smaller multiplier for the RTO exponential back off and randomized multiplier for the RTO exponential back off were unhelpful. The reason is that there are only a tiny number of exponential back offs for the entire transfer. Thus, altering the exponential back off behavior had little impact. We saw this phenomenon most clearly for the default 200ms minimum RTO timer value. Figure 3 shows a histogram for the inter-packet send times for an entire transfer. The bins greater than 200ms are nearly empty.

Surprisingly, randomizing the minimum and initial RTO timer value was also unhelpful. This was against our intuitions. However, this is what we observed in Figure 4. Randomizing the minimum and initial RTO timer does not improve goodput and it does not impose a goodput penalty. A possible explanation is that the switch buffer is a fundamentally shared resource, so regardless of any randomization at the senders, whenever the switch buffer gets filled, it is filled for all senders at the same time. The senders may restart their transmissions at random times, but all subsequent switch buffer overflow events will be synchronized for all senders because the senders fundamentally share the same switch.

[1]Due to space considerations we omit the code snippets.

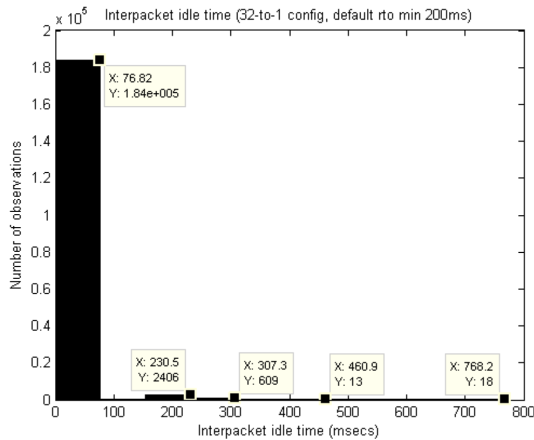

Figure 3: Inter-packet idle time histogram

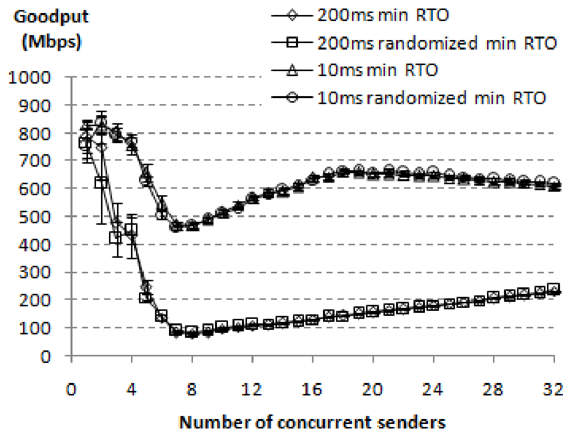

Figure 4: Comparison between randomized and non-randomized minimum/initial RTO values

TCP delayed ACKs were not disabled in the environment for these measurements. We will have a more detailed discussion regarding delayed ACKs later in the paper.

We found that the most promising modification was to reduce the minimum RTO timer value. We used this modification as a reference point for developing an in-depth understanding of the Incast phenomenon.

5. ANALYSIS IN DEPTH

5.1 Different RTO Timers

We ran the fixed-fragment transfer workload for several different minimum RTO timer values. The results are in Figure 5. We observed three distinct regions for up to 48 concurrent senders, labeled R1, R2 and R3 for the RTO min 1ms curve. The initial throughput collapse (R1) is followed by a region of goodput increase (R2), which is followed by yet another turning point and a region of goodput decrease (R3). For different minimum RTO timer values, the location of the turning points and the slope between turning points are different. For the default minimum RTO of 200ms, the second turning point seems to be beyond 48 nodes.

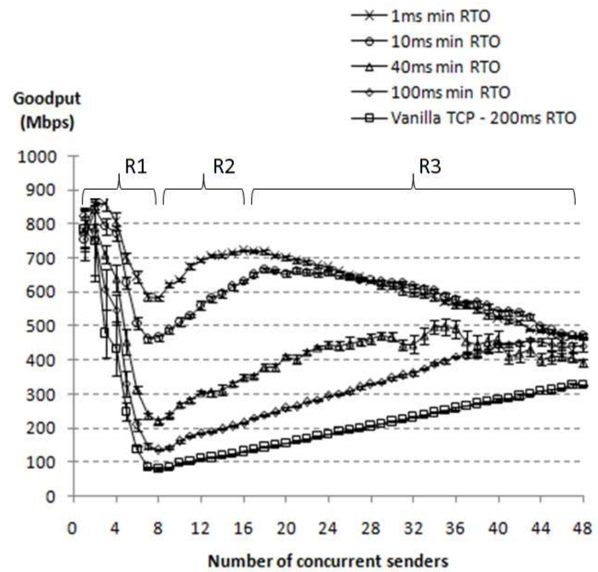

Figure 5: Goodput for various minimum RTO values

There are several other noteworthy details. Smaller minimum RTO timer values mean larger values for the initial goodput minimum. Also, the initial goodput minimum occurs at the same number of senders, regardless of minimum RTO timer values. Furthermore, larger minimum RTO timer values mean the goodput local maximum occurs at a larger number of senders. Smaller minimum RTO timer values mean faster goodput "recovery" between the initial goodput minimum and the subsequent goodput local maximum. After the goodput local maximum, the slope of goodput decrease is the same for all minimum RTO timer values. Ideally, a model should explain both the general shape of the curve, and account for these details.

5.2 Delayed ACKs and High Resolution Timers

Two mechanisms proposed in [11] include modifying the operating system to use high resolution timers, which facilitates RTO timers with the granularity of hundreds of microseconds. Another suggestion was to turn off delayed ACKs wherever possible, such that the delay ACK threshold does not act as an unnecessary timeout. Most existing TCP stacks implement delayed ACKs by default. In Linux the delayed ACKs threshold is 40ms. Hence disabling delayed ACKs is expected to improve performance for RTO timer values of 40ms or less. The authors of [11] shared with us their Linux kernel modifications, and we repeated their experiments for their original workload (the *fixed-fragment workload*), with vastly different and somewhat unexpected results.

Our results are in Figure 6. Some goodput numbers there are slightly different from the numbers for the same experiments in Figure 5. The reason is the high resolution timers kernel patch was built for Linux 2.6.28.1, whereas the data in Figure 5 is for Linux 2.6.18.8.

To ensure a rigorous comparison between high and low resolution timers, we perform the measurements in Figure 6 for Linux 2.6.28.1.

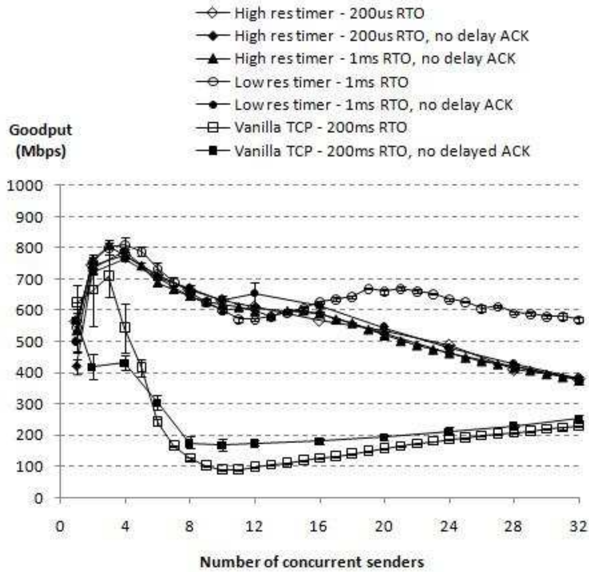

Figure 6: Disabling delayed ACKs and using high resolution timers

In Figure 6, the solid markers indicate experiment settings with delayed ACKs disabled, and the hollow markers indicate experiments in which delay ACKs were not disabled. For vanilla TCP with an RTO timer value of 200ms, disabling delay ACKs lead to a slight improvement. For an RTO timer value of 1ms, disabling delayed ACKs actually creates a significant performance penalty. The kernel modification for the high resolution timer has delayed ACKs disabled by default. Using a high resolution RTO timer of 1ms or 200μs give the same performance as that of a low resolution RTO timer of 1ms with delayed ACKs disabled. Surprisingly, high resolution RTO timer of 200μs with delayed ACKs enabled gives near identical performance to RTO 200μs with delayed ACKs disabled. Out of the experimental settings we compared, the low resolution RTO timer of 1ms with delayed ACKs turned on is actually optimal.

This result is most unexpected and a departure from the findings in [11]. Delayed ACKs were originally meant to prevent ACK congestion. In the datacenter network environment, we believe that it is unlikely for ACK congestion to occur, even with delayed ACKs turned off. The remaining plausible explanation would be that disabling delayed ACKs would cause a rapid stream of ACK arrivals at the data sender, resulting in the sender over-driving its congestion window. In steady state, the TCP congestion window would be able to adapt, but the high bandwidth-delay product of the round trip path, means that TCP would never reach steady-state before a fragment has been transmitted.

It turns out that our hypothesis with regard to ACK over-driving the TCP congestion window is correct. We looked at the TCP internal state variables using the TCP info socket option. Figures 7 and 8 compare the congestion window behavior for TCP connection with and without delayed ACKs, for the 8-to-1 configuration for low resolution 1ms RTO values. We see that with delayed ACKs turned off, the congestion window exhibits larger fluctuations and a higher average. If the lower and more stable congestion window behavior is already associated with a partial goodput collapse, then the higher and less stable congestion window will definitely correspond to a more severe goodput collapse.

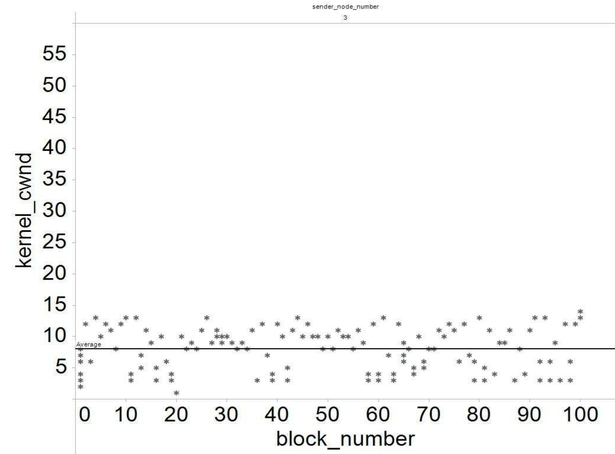

Figure 7: Congestion window with delay ACKs

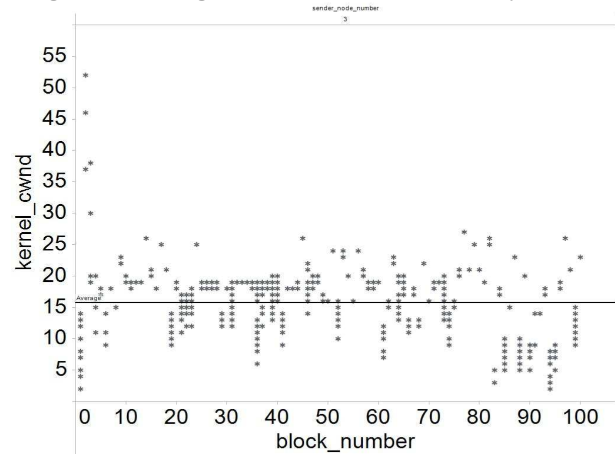

Figure 8: Congestion window without delay ACKs

The sub-optimal behavior of disabling delayed ACKs is correlated with an increased number of smoothed RTT spikes, as evident in Figures 9 and 10, showing the smoothed RTT comparison for the 8-to-1 configuration for low resolution 1ms RTO values. The average smoothed RTT remains the same with or without delayed ACKs, suggesting that the underlying network conditions did not change. Thus, the increased number of smoothed RTT spikes represent more frequent, unnecessary congestion timeout events, another piece of evidence that indicates that the congestion window is being over-driven when delay ACKs are turned off.

5.3 Workload and Testbed

It remains to be explained why the choice of workload affects the results so much, or whether the sub-optimal performance of disabling delayed ACKs is caused by a different

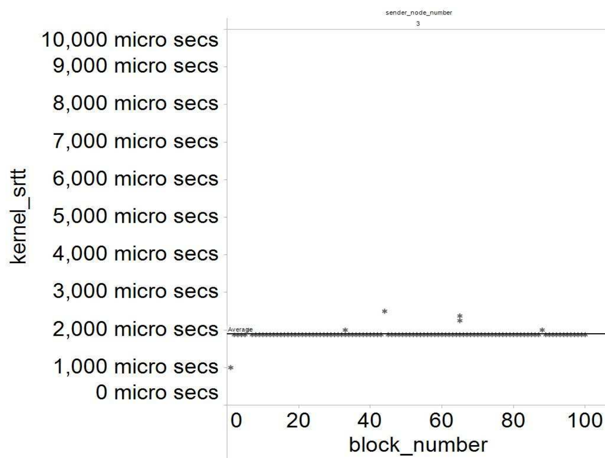

Figure 9: Smoothed RTT with delayed ACKs

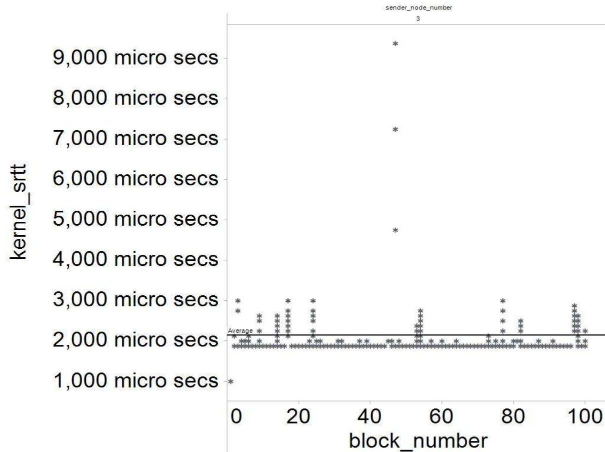

Figure 10: Smoothed RTT without delayed ACKs

experimental environment. To clarify our own uncertainty, we repeated the variable-fragment workload in [11] on our testbed.

Our results are in Figure 11. There are quite a few interesting phenomena worthy of highlighting. Most notably, for the low resolution timers, the variable-fragment workload produces very simple goodput behavior - initial goodput collapse followed by a flat goodput line. This is a significant contrast to the complex behavior in Figure 5. We believe the variable-fragment workload does not fully reflect the complex dynamics of the incast pathology. The initial goodput collapse is still evident. However, the subsequent complex dynamics are hidden. In a sense, as we scale up to an increasing number of senders, the variable-fragment workload is placing the same level of stress on the network, because the total amount of data to be transmitted in each fragment is the same, regardless of the number of senders. In comparison, the fixed-fragment workload places an increasing level of stress on the network as we scale up to more senders. The fragment size is the same, so with more senders, we transmit a larger sum of data. The two workloads place two fundamentally different stress patterns on the network, thus the response from the network should be different also.

More importantly, the goodput behavior for high resolution timers with 200us RTO value is similar to the behavior in Figure 6, and different from the findings of [11]. The results here tell us several things. First, whatever sub-optimal behavior we see with regard to delayed ACKs is workload independent. Second, because our results are different from [11], and we ensured that all application and transport level parameters are reproduced, the different results suggest that the different network environment associated with a different testbed would play a part. Also, in agreement with the observation in Figure 6, the sub-optimal behavior for 200us RTO value is independent of the presence or absence of delayed ACKs.

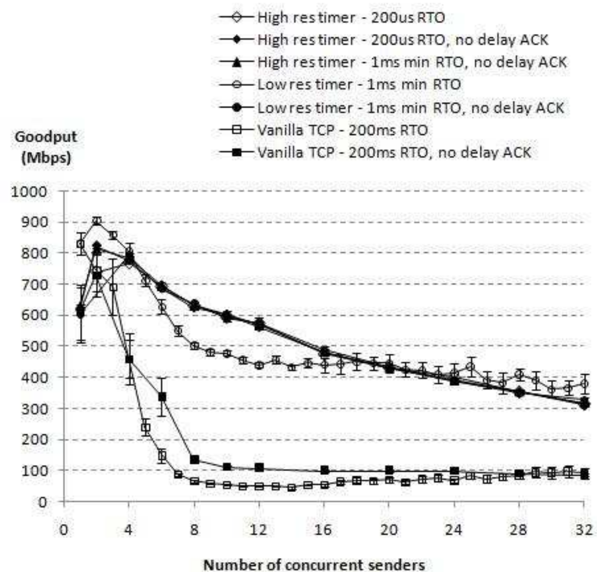

Figure 11: Variable fragment size workload

There is actually a nuanced explanation that accounts for all the observed phenomena. We already know that the absence of delayed ACKs overdrives the congestion window, leading to workload-independent, sub-optimal behavior for TCP that has delay ACKs disabled. The 200us RTO value is network dependent, and delayed ACKs independent. The explanation lies in the smoothed RTT values we measured in Figure 10. There, we see that the average smoothed RTT values are approximately 2ms, regardless of the amount of smoothed RTT spikes. This means that the round trip time for our network, at least as interpreted by the TCP stack and stored as the smoothed RTT TCP protocol state variable, is approximately 2ms. Thus, a RTO value of 200us would be a severe mismatch with the perceivable RTT measured by the TCP stack. This mismatch is independent of the absence or presence of delayed ACKs. Overdriving congestion window and a mismatch between RTO and RTT would both result in a significantly larger number of spurious retransmissions. The number of spurious retransmissions in both cases would be bounded by the speed at which packets can be delivered and drained from the network. The network has a fixed limit at which it can drain packets, independent of the transport mechanisms used. Thus, the phenomenon produced by the two mechanisms is the same, even though the causes are different. Hence, the 1ms RTO no delayed

ACKs TCP exhibits identical behavior to 200us RTO with or without delayed ACKs, independent of workload, but differs from the findings in [11] because of a potential difference in the network round trip time. This completes a qualitative explanation of all the different findings between our experiments and those in [11].

We can quantitatively verify our accounting here by measuring the number of spurious retransmissions. We can extract the data from the tcpdumps that we already collect. However, due to time constraints, we have not yet performed this analysis.

What remains is for us to provide a satisfactory explanation for the complex behavior observed in the fixed-fragment workload, as evident in Figure 5. In the next section, we outlines the first steps we have taken towards a helpful analytical model.

6. QUANTITATIVE MODELS

6.1 Model Description

We took the most promising results for our fixed-fragment workload, i.e., delay ACKs not disabled, low resolution timers, and developed a relatively simple model that partially accounts for the general shape of the curve, and predicts the numerical goodput values within an empirical scaling factor. As a first step, we focus on the relatively simple curve for 200ms minimum RTO timer value.

To the first order, the goodput of each sender is given by Equation 1. D is the total amount of data to be sent, 100 blocks of 256KB each. L is the total transfer time of the workload without any RTO events. R is the number of RTO events during the transfer, and r is the value of the minimum RTO timer value. Equation 1 captures the intuition that data transfer rate is the data transfer size divided by the data transfer time. There are very few exponential RTO back offs, so r approximates the effective timeout duration of all RTO events. The net goodput across S senders is Equation 2.

$$\frac{D}{L+(R*r)} \tag{1}$$

$$\frac{S*D}{L+(R*r)} \tag{2}$$

For a particular curve, D and r are constant, while L and R may be functions of S. We could not make assumptions about how they vary with S. However, we were able to fit piece-wise linear function to the empirically observed behavior. The observed variation between R and S is shown in Figure 12. We used Equation 3 to fit the curve.

$$R = \begin{cases} (\frac{35}{10})*S & : \quad S <= 10 \\ 35 & : \quad S > 10 \end{cases} \tag{3}$$

Quantifying the variation between L and S is more challenging. We could not just use the measured overall transfer time, because that would include the time spent in RTO. Instead, we used D divided by the link bandwidth as a baseline for L, and add to $L_{baseline}$ the number of packets multiplied by the time between successive packet transfers. The intuition is that without RTOs, L would be increased by a longer average inter-packet wait time. The inter-packet wait time is empirically observable, and expected to vary with S. In symbolic form, we compute L according to Equation 4, where I is the inter-packet wait time, and the number of packets is given by D divide by the average MSS. From empirically observed data, we approximate average MSS size to be 1500 bytes, close to the minimum MSS size of 1448 bytes for the DETER testbed.

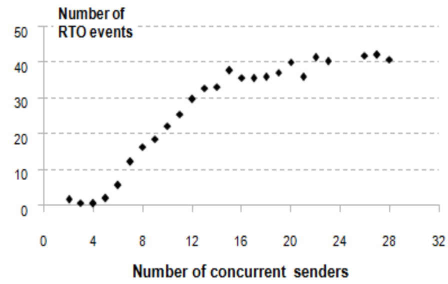

Figure 12: RTO counts per sender 200ms minimum RTO

$$L = \frac{D}{Bandwidth} + \left(\frac{D}{averageMSS}\right)*I \tag{4}$$

The empirically observed variation between I and S is shown in Figure 13. Again, we fit a piece-wise linear function to the observed data, as in Equation 5. We also note with interest that the variation between I and S is nearly identical to that between R and S.

$$I = \begin{cases} (\frac{4.5}{10})*S & : \quad S <= 10 \\ 4.5 & : \quad S > 10 \end{cases} \tag{5}$$

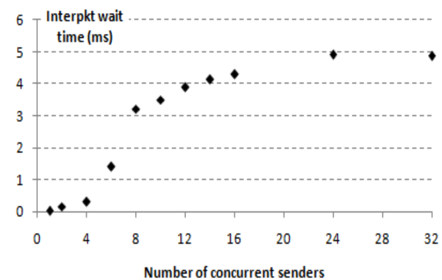

Figure 13: Average inter-packet idle time for 200ms minimum RTO, selected data points

Substituting L and R into Equation 2 yields Figure 14, showing the predicted and the measured goodput graphs for the default 200ms minimum RTO timer. The shapes of the curves are identical, and the numerical goodput values agree with each other within an experimental scaling factor. Later, we give a qualitative explanation for the goodput underestimate.

This model accounts for the shape of the initial goodput collapse and the goodput "recovery". However, it is not yet a complete model. It does not account for the second order goodput decrease. It also fails to predict the measured

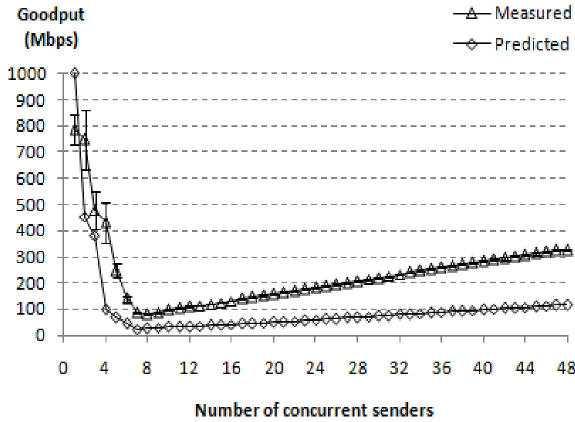

Figure 14: Predicted and measured goodput for 200ms minimum RTO

curves for small minimum RTO timer values. Figure 15 shows the predicted and measured curves for the 1ms minimum RTO timer. Figure 16 shows the measured variation in R and I for this modified kernel, together with the corresponding measurements for 200ms minimum RTO timer. Most notably, the variation between R and S is similar for different minimum RTO timer values, but the variation between I and S is vastly different. Also, for the 1ms minimum RTO timer, the values of the inter-packet wait time is comparable and even greater than the RTO timer value.

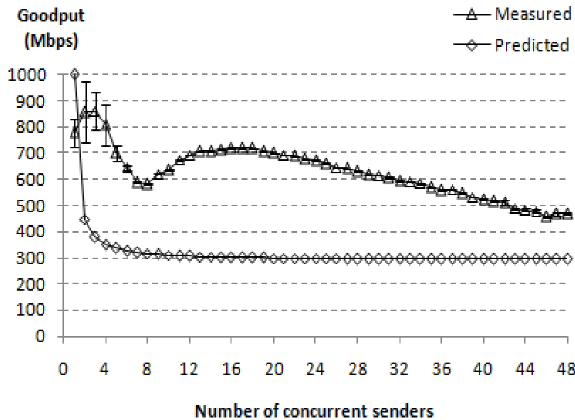

Figure 15: Predicted and measured goodput for 1ms minimum RTO

This model offers us two critical insights into potential TCP fixes to help address the Incast problem. The goodput is heavily affected by both the minimum RTO timer value and the inter-packet wait time. For large RTO timer values, reducing the RTO timer value is a first-order mitigation. For smaller RTO timer values, intelligently controlling the inter-packet wait time becomes crucial.

The incompleteness of this model compelled us to develop a more complex construction to explain the other features

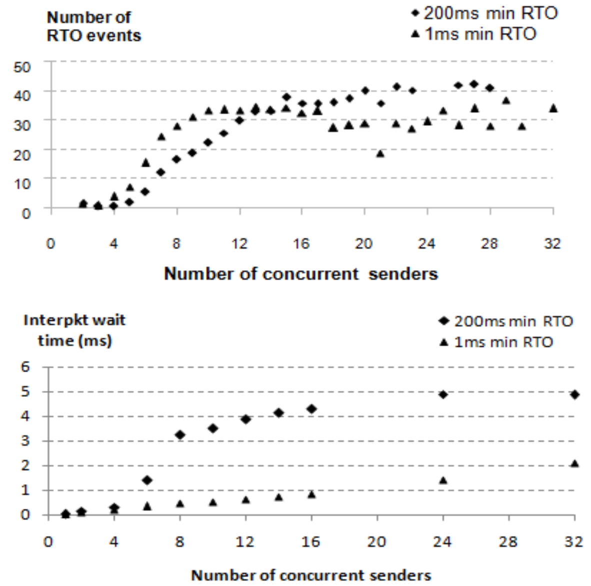

Figure 16: RTO counts and average inter-packet idle time for 1ms and 200ms min RTO

in the goodput graph. These model refinements are outlined next. We have not yet numerically verified that model against experimental data in a manner that is mathematically rigorous. However, it offers plausible explanations for almost all the observed features in the goodput graphs.

6.2 Qualitative Refinements

There are several elements to the refinements:

1. As the number of senders increase, the number of RTO events per sender increases. Beyond a certain number of senders, the number of RTO events is constant.

   Justification: This is the behavior observed in and Figure 16. We intuitively expect that more senders would cause more congestion, leading to a large number of RTO events. We also expect that when RTO events are frequent, TCP would be able to adapt and cap the frequency of RTO events.

2. When a network resource becomes saturated, it is saturated at the same time for all senders.

   Justification: We intuitively expect this - all senders share the bottleneck network resource, i.e., the switch buffers. When this resource is saturated, it is saturated for all senders that share the resource.

3. After a congestion event, the senders enter the TCP RTO state. The RTO timer expires at each sender with a uniform distribution in time and a constant delay after the congestion event, i.e., the probability that a given sender's RTO timer expires at time t after a congestion event is:

$$Prob(\text{enter RTO at t}) = \begin{cases} \frac{1}{T} & : \quad d < t < d + T \\ 0 & : \quad otherwise \end{cases} \tag{6}$$

d is the delay for congestion info to propagate back to the sender, which is expected to be nearly negligible in the datacenter. T is the width of the uniform distribution in time.

Justification: We expect d to exist because there is delay in the network, however small it may be in the datacenter. We assume uniform distribution because we expect packets from different senders to be evenly dispersed in the network buffer. Even if the packets from different senders arrive in bursts or groups, we expect the bursts and groups to be evenly distributed. We expect the uniform distribution to be located between d and $d + T$ because a sender would enter TCP RTO state when the first unsuccessfully transmitted packet times out. This would occur after the ACK for the last successfully transmitted packet arrives at the sender. The last ACK is expected to arrive at time d + some uniformly distributed t. The value of t would depend on the position of the last successful packet from a given sender relative to the last packets from other senders.

4. T increases as the number of senders increases. However, T is bounded.

   Justification: We expect T to have these properties because the more senders there are, the larger would be the batch of last packets. The larger the batch of last packets there are, the greater the variation in the time to drain the last packet from a particular sender. We expect T to be bounded because the network buffers are finite.

Using these refinements, we can explain the three distinct regions in the goodput graph in Figure 5 as follows.

Region 1: Initial goodput collapse.

Explanation: The number of RTO events increases and the number of senders increases. This is already accounted for this effect earlier in Section 6.1.

Region 2: Goodput recovery.

Explanation: This region was also accounted for earlier. The construction here refines the earlier model as follows. As the number of senders increase, T increases, and there is less overlap in the RTO periods for different senders. This means the impact of RTO events is less severe - a mitigating effect. Consequently, the effective value of the RTO timer would be reduced as the number of senders increases. We believe this effect partially accounts for the goodput underestimation in Figure 14.

Region 3: Goodput decreases again.

Explanation: This region was not accounted for earlier. As the number of senders increases, T would eventually become comparable or even larger than the value of RTO timers. When this happens, there would be interference between senders retransmitting after an RTO timeout and senders that are transmitting because they have not yet entered the RTO state. This is an exacerbating effect. A possible consequence is increased inter-packet wait time, leading to the behavior in Figure 16, and a gradual decrease in goodput.

Combined with our earlier model, the refinements here can explain many details in the goodput graph.

Detail 1: A smaller minimum RTO timer value means larger goodput values for the initial minimum.

Explanation: The default TCP RTO timer value is much larger than the network round trip time in datacenters. Hence, smaller RTO timers mean that the impact of RTO events is less severe. This explanation is unchanged from Section 6.1.

Detail 2: The initial goodput minimum occurs at the same number of senders, regardless of the value of the minimum RTO timer.

Explanation: The initial goodput minimum roughly corresponds to the transition point in R in Figure 16. The transition point is at roughly the same number of senders for all RTO timer values. Hence the goodput minimum is roughly the same for all RTO timer values. This explanation is unchanged from Section 6.1.

Detail 3: The second order goodput peak occurs at a higher number of senders for a larger RTO timer value.

Explanation: The second order maximum corresponds to the trade-off between incremental goodput increases with more senders and increases of T causing interference with retransmissions. The cross over point depends on both the number of senders and the RTO timer value. In particular, larger RTO timer values would require larger T for the effects to cross over. If the increase in T depends on the number of senders only, then the second order goodput peak would occur at more senders for larger RTO timer values. This explanation was missing earlier.

Detail 4: The smaller the RTO timer values, the faster the rate of recovery between the goodput minimum and the second order goodput maximum.

Explanation: For smaller RTO timer values, the same increase in T will have a larger mitigating effect. Hence, as the number of senders increases, the same increase in T will result in a faster increase in the goodput for smaller RTO timer values. This explanation was missing earlier.

Detail 5: After the second order goodput maximum, the slope of goodput decrease is the same for different RTO timer values.

Explanation: When T becomes comparable or larger than the RTO timer value, the amount of interference between retransmits after RTO and transmissions before RTO no longer depends on the value of the RTO timer. The amount of interference increases with the number of senders at a fixed rate, leading to a fixed rate of goodput degradation after the second order goodput maximum. This explanation was missing earlier.

The model refinements here have qualitatively accounted for many details in the goodput graphs. However, it is not yet a complete model, because we have not yet quantitatively justified its refinements, nor numerically verified these refinements against experimentally measured data. The success criteria for the refinements here would be to quantitatively re-create the shape and trends of the measured goodput curves, as we have done more rigorously in the previous section.

7. CONCLUSIONS AND FUTURE WORK

In this paper we studied the dynamics of Incast. To develop a better understanding of the phenomenon, we conducted experiments on a configurable network testbed, enabling fine-grained control over end hosts and the network. Based on analysis of empirical data, we account for the difference between our observations and that in related work. We also propose a simple mathematical model to explain some of the observed trends.

For future work, we plan to extend the model to quantitatively account for all goodput trends. We have gathered a large collection of TCP trace data for a variety of protocol settings. We want to identify principled fixes to Incast that are general across workloads and network environments. This implies that the TCP protocol that addresses Incast would be a more general and robust transport protocol than what TCP is today. The first step in this design process would be to narrow down the range of TCP variables of interest. Some variables are inter-dependent with others, some variables may have no impact on goodput at all. The abundance of somewhat counter-intuitive findings we encountered suggests to us that intuitive analysis alone is insufficient. Thus, we plan to employ machine learning to help us identify the most important control variables to examine. Then, the breadth of our analysis may be significantly reduced, and we would be able to focus on a handful of parameters instead of a horde of parameters.

Also, the mismatch between RTO and RTT, and the complexity with regard to delayed ACKs, are both fundamentally associated with the ACK-clocked nature of the TCP protocol. If we are able to identify a small set of core parameters of interest, we may also be able to construct correlations between the parameters. These correlations, in turn, may allow us to develop control models that fundamentally depart from the ACK-clocked nature of TCP. Such a transport level protocol may be able to survive conditions in which ACK-clocked transport fails.

Lastly, our analysis with regard to different workloads and different environments highlighted for us the necessity of evaluating any proposed solutions under a wide variety of settings. Thus, to validate whatever fixes we propose, we also plan to evaluate our mechanisms for different applications, environments, network equipment, and network topologies. We will investigate the associated tradeoffs, and evaluate our solutions using real life workloads. In the long term, we will also use our understanding to construct a detection signature or benchmark that would be used to assess whether specific applications or workloads are affected by Incast.

8. ACKNOWLEDGMENTS

The authors would like to thank Lifan Zhang, Jon Kuroda and Keith Sklower for their help and support. This research is supported in part by gifts from Sun Microsystems, Google, Microsoft, Cisco Systems, Hewlett-Packard, IBM, Network Appliance, Oracle, Siemens AB, and VMWare, and by matching funds from the State of California's MICRO program (grants 06-152, 07-010, 06-148, 07-012, 06-146, 07-009, 06-147, 07-013, 06-149, 06-150, and 07-008), the National Science Foundation (grant #CNS-0509559), and the University of California Industry/University Cooperative Research Program (UC Discovery) grant COM07-10240.

9. REFERENCES

[1] tcpdump. http://www.tcpdump.org/.

[2] tcptrace. http://irg.cs.ohiou.edu/software/tcptrace/.

[3] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *SIGCOMM '08: Proceedings of the ACM SIGCOMM 2008 conference on Data communication*, pages 63–74, New York, NY, USA, 2008. ACM.

[4] S. Bellovin. Defending against sequence number attacks (rfc 1948). http://www.ietf.org/rfc/rfc1948.txt.

[5] T. Benzel, R. Braden, D. Kim, C. Neuman, A. Joseph, K. Sklower, R. Ostrenga, and S. Schwab. Design, deployment, and use of the deter testbed. In *DETER: Proceedings of the DETER Community Workshop on Cyber Security Experimentation and Test on DETER Community Workshop on Cyber Security Experimentation and Test 2007*, pages 1–8, Berkeley, CA, USA, 2007. USENIX Association.

[6] D.-M. Chiu and R. Jain. Analysis of the increase and decrease algorithms for congestion avoidance in computer networks. *Comput. Netw. ISDN Syst.*, 17(1):1–14, 1989.

[7] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[8] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, 2003.

[9] V. Jacobson. Congestion avoidance and control. In *SIGCOMM '88: Symposium proceedings on Communications architectures and protocols*, pages 314–329, New York, NY, USA, 1988. ACM.

[10] A. Phanishayee, E. Krevat, V. Vasudevan, D. G. Andersen, G. R. Ganger, G. A. Gibson, and S. Seshan. Measurement and analysis of TCP throughput collapse in cluster-based storage systems. In *FAST'08: Proceedings of the 6th USENIX Conference on File and Storage Technologies*, pages 1–14, Berkeley, CA, USA, 2008. USENIX Association.

[11] V. Vasudevan, A. Phanishayee, H. Shah, E. Krevat, D. G. Andersen, G. Ganger, G. A. Gibson, and B. Mueller. Safe and Effective Fine-grained TCP Retransmissions for Datacenter Communication. In *SIGCOMM '09: Proceedings of the ACM SIGCOMM 2009 conference on Data communication*, 2009.