Safe and Effective Fine-grained TCP Retransmissions for Datacenter Communication

Vijay Vasudevan[1], Amar Phanishayee[1], Hiral Shah[1], Elie Krevat[1],
David G. Andersen[1], Gregory R. Ganger[1], Garth A. Gibson[1,2], Brian Mueller[2]

[1]Carnegie Mellon University, [2]Panasas Inc.

ABSTRACT

This paper presents a practical solution to a problem facing high-fan-in, high-bandwidth synchronized TCP workloads in datacenter Ethernets—the TCP incast problem. In these networks, receivers can experience a drastic reduction in application throughput when simultaneously requesting data from many servers using TCP. Inbound data overfills small switch buffers, leading to TCP timeouts lasting hundreds of milliseconds. For many datacenter workloads that have a barrier synchronization requirement (e.g., filesystem reads and parallel data-intensive queries), throughput is reduced by up to 90%. For latency-sensitive applications, TCP timeouts in the datacenter impose delays of hundreds of milliseconds in networks with round-trip-times in microseconds.

Our practical solution uses high-resolution timers to enable microsecond-granularity TCP timeouts. We demonstrate that this technique is *effective* in avoiding TCP incast collapse in simulation and in real-world experiments. We show that eliminating the minimum retransmission timeout bound is *safe* for all environments, including the wide-area.

Categories and Subject Descriptors

C.2.2 [**Computer-Communication Networks**]: Network Protocols—*TCP/IP*; C.2.6 [**Computer-Communication Networks**]: Internetworking

General Terms

Performance, Experimentation, Measurement

Keywords

Datacenter Networks, *Incast*, Performance, Throughput

1. INTRODUCTION

In its 35 year history, TCP has been repeatedly challenged to adapt to new environments and technology. Researchers have proved adroit in enabling TCP to function well in gigabit networks [27], long/fat networks [18, 8], satellite and wireless environments [22, 5], among others. In this paper, we examine and improve performance in an area that, surprisingly, also proves challenging to TCP: very low delay, high throughput, datacenter networks of dozens to thousands of machines.

The problem we study is TCP incast collapse [25], where application throughput drastically reduces when multiple senders communicate with a single receiver in high bandwidth, low delay networks using TCP. Highly bursty, fast data transmissions overfill Ethernet switch buffers, causing intense packet loss that leads to TCP timeouts. These timeouts last hundreds of milliseconds on a network whose round-trip-time (RTT) is in the 10s or 100s of microseconds. Protocols that have some form of synchronization requirement, such as filesystem reads and writes or highly parallel data-intensive queries found in large `memcached` clusters [12], block waiting for timed-out connections to finish before issuing new requests. These timeouts and the resulting delay can reduce application throughput by 90% (Section 3) or more [25, 28]. Coarse-grained TCP timeouts can also harm performance for latency-sensitive datacenter applications (Section 2.2).

In this paper, we present and evaluate a set of system extensions to enable microsecond-granularity retransmission timeouts (RTO). The challenges in doing so are threefold: First, we show that the solution is *practical* by modifying the Linux TCP implementation to use high-resolution kernel timers. Second, we show that these modifications are *effective*, preventing TCP incast collapse in a real cluster for up to 47 concurrent senders (Section 5). As shown in Section 4, microsecond granularity timeouts are necessary—simply reducing RTO_{min} in today's TCP implementations without also improving the timing granularity does not prevent TCP incast collapse, particularly in future settings. In simulation, our changes to TCP prevent incast collapse for up to 2048 concurrent senders on 10-gigabit links. Last, we show that the solution is *safe*, examining the effects of an aggressively reduced RTO in the wide-area, showing that its benefits to recovery in datacenters do not affect performance for bulk flows in the wide-area.

The motivation for solving this problem is the increasing interest in using Ethernet and TCP for communication and bulk storage transfer applications in the fastest, largest datacenters. Provided that TCP adequately supports high bandwidth, low latency, synchronized and parallel applications, there is a strong desire to "wire-once" and reuse the mature, well-understood transport protocols that are so familiar in lower bandwidth networks.

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.
SIGCOMM'09, August 17–21, 2009, Barcelona, Spain.
Copyright 2009 ACM 978-1-60558-594-9/09/08 ...$5.00.

Scenario	RTT	OS	TCP RTO_{min}
WAN	100ms	Linux	200ms
Datacenter	<1ms	BSD	200ms
SAN	<0.1ms	Solaris	400ms

Table 1: Typical round-trip-times and minimum TCP retransmission bounds.

2. BACKGROUND

Cost pressures increasingly drive datacenters to adopt commodity components, and often low-cost implementations of such. An increasing number of clusters are being built with off-the-shelf rackmount servers interconnected by Ethernet switches. While the adage "you get what you pay for" still holds true, entry-level gigabit Ethernet switches today operate at full data rates, switching upwards of 50 million packets per second—at a cost of about $10 per port. Commodity 10Gbps Ethernet is now cost-competitive with specialized interconnects such as Infiniband and FibreChannel, and benefits from wide "brand recognition". To reduce cost, however, lower-cost switches often sacrifice expensive, power-hungry SRAM packet buffers, the effect of which we explore throughout this work.

The desire for commodity parts extends to transport protocols. TCP provides a "kitchen sink" of protocol features, including reliability via retransmission, congestion and flow control, and in-order packet delivery to the receiver. Not all applications need all of these features [20, 31], and some benefit from more rich transport abstractions [14], but TCP is mature and well-understood by developers, and has become the transport protocol of choice even in many high-performance environments.

Without link-level flow control, TCP is solely responsible for coping with and avoiding packet loss in the (often small) Ethernet switch egress buffers. Unfortunately, the workload we examine has three features that challenge (and nearly cripple) performance: a highly parallel, synchronized request workload; buffers much smaller than the bandwidth-delay product of the network; and high-fan-in communication resulting in TCP flows with windows of only a few packets.

2.1 TCP Incast Collapse

Barrier-synchronized request workloads are becoming increasingly common in today's commodity clusters. Examples include parallel reads/writes in cluster filesystems such as Lustre [6], Panasas [34], or NFSv4.1 [33]; search queries sent to dozens of nodes, with results returned to be sorted[1]; or parallel databases that harness multiple back-end nodes to process parts of queries. We define the request to be "barrier synchronized" when the client cannot make forward progress until the responses from every server for the current request have been received—these applications often cannot present partial results or issue an unbounded number of requests.

In a cluster file system, for example, a client application requests a data block striped across several storage servers, issuing the next data block request only when all servers have responded with their portion. This workload can result in packets overfilling the buffers on the client's port on the switch, resulting in many losses. Under severe packet loss, TCP can experience a timeout that lasts a minimum of 200ms, determined by the TCP minimum retransmission timeout (RTO_{min}). While the default values operating systems use today may suffice for the wide-area, datacenters and SANs have round-trip-times that are orders of magnitude below the RTO_{min} defaults (Table 1).

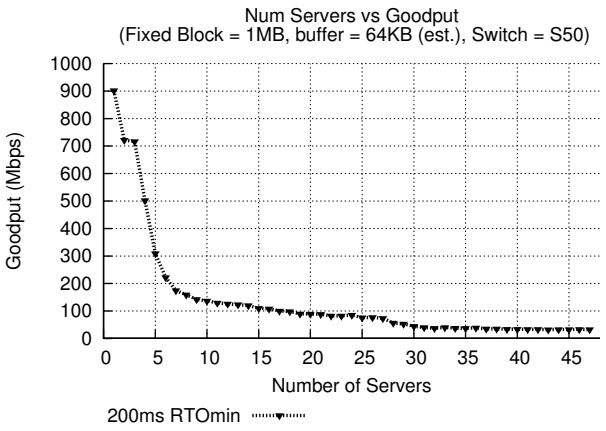


Figure 1: TCP incast collapse is observed for a synchronized reads application running on a 48-node cluster

When a server involved in a barrier-synchronized request experiences a timeout, other servers can finish sending their responses, but the client must wait a minimum of 200ms before receiving the remaining parts of the response, during which the client's link may be completely idle. The resulting throughput seen by the application may be as low as 1-10% of the client's bandwidth capacity.

Figure 1 shows the throughput of our test synchronized-read application (Section 3.2) as we increase the number of nodes it reads from, using an unmodified Linux TCP stack. This application performs synchronized reads of 1MB blocks of data; that is, each of N servers responds to a block read request with 1 MB / N bytes at the same time. Even using a high-performance switch (with its default settings), the throughput drops drastically as the number of servers increases, achieving only 3% of the network capacity when it tries to stripe the blocks across all 47 servers.

To summarize, the preconditions for TCP incast collapse are as follows:

1. High-bandwidth, low-latency networks with small switch buffers;
2. Clients that issue *barrier-synchronized* requests in parallel: the client does not issue new requests until all responses from the current request have been returned;
3. Servers that return a relatively small amount of data per request

If precondition 2 does not hold, then a timed out flow does not stall the client from making forward progress on other flows and hence will continue to saturate the client's link. If precondition 3 does not hold and at least one flow is active at any time, the active flows will have enough data to send to saturate the link for 200ms—until the stalled flows retransmit and continue.

[1]In fact, engineers at Facebook recently rewrote the middle-tier caching software they use—memcached [23]—to use UDP so that they could "implement application-level flow control for ... gets of hundreds of keys in parallel" [12]

2.2 Latency-sensitive Applications

While the focus of this work is on the throughput collapse observed for synchronized reads and writes, the imbalance between the TCP RTO_{min} and datacenter latencies can result in poor performance for applications sensitive to millisecond delays in query response time. In an interactive search query where a client requests data from dozens of servers in parallel, any flow that experiences a timeout will be delayed by 200ms. If the client cannot make forward progress (i.e., present results to the user) until all results are received, the entire request will be stalled for a minimum of 200ms, resulting in poor query latency.

To demonstrate this, we performed the following experiment: we started ten bulk-data transfer TCP flows from ten clients to one server. We then had another client issue small request packets for 1KB of data from the server, waiting for the response before sending the next request. Approximately 1% of these requests experienced a TCP timeout, delaying the response by at least 200ms. Even without incast communication patterns, a latency-sensitive application can observe TCP timeouts due to congested queues caused by cross-traffic. The fine-grained TCP retransmission techniques we use to prevent TCP incast collapse will also benefit these more responsive latency-sensitive applications.

2.3 Prior Work

The TCP incast problem was first termed "Incast" and described by Nagle *et al.* [25] in the context of parallel filesystems. Nagle *et al.* coped with TCP incast collapse in the parallel filesystem with application-specific mechanisms. Specifically, Panasas [25] limits the number of servers simultaneously sending to one client to about 10 by judicious choice of the file striping policies. They also cap the advertised window size by reducing the default size of per-flow TCP receive buffers on the client to avoid incast collapse on switches with small buffers. For switches with large buffers, Panasas provides a mount option to increase the client's receive buffer size. In contrast, this work provides a TCP-level solution for switches with small buffers and many more than 10 simultaneous senders that does not require implementing application-specific mechanisms. Also, our solution does not require re-implementing the many features of TCP within a UDP framework, perhaps as was the case with Facebook [12].

Prior work characterizing TCP incast collapse ended on a somewhat down note, finding that TCP improvements—NewReno, SACK [22], RED [13], ECN [30], Limited Transmit [1], and modifications to Slow Start— sometimes increased throughput, but did not substantially prevent TCP incast collapse because the majority of timeouts were caused by full window losses [28]. This work found three partial solutions: First, larger switch buffers could delay the onset of incast collapse (doubling the buffer size doubled the number of servers that could be contacted), but at substantial dollar cost. Second, Ethernet flow control was effective when the machines were on a single switch, but was dangerous across inter-switch trunks because of head-of-line blocking. Finally, reducing TCP's minimum *RTO*, in simulation, appeared to allow nodes to maintain high throughput with several times as many nodes—but was left unexplored because of practical implementation concerns with microsecond timeouts. In this paper, we address the practicality, effectiveness and safety of very short timeouts in depth.

3. EVALUATING THROUGHPUT WITH FINE-GRAINED RTO

How low must the *RTO* be to retain high throughput under TCP incast collapse conditions, and to how many servers does this solution scale? We explore this question using real-world measurements and ns-2 simulations [26], finding that to be maximally effective, the timers must operate on a granularity close to the RTT of the network—hundreds of microseconds or less.

3.1 Jacobson RTO Estimation

The standard *RTO* estimator [17] tracks a smoothed estimate of the round-trip time, and sets the timeout to this RTT estimate plus four times the linear deviation—roughly speaking, a value that lies outside four standard deviations from the mean:

$$RTO = SRTT + (4 \times RTTVAR) \quad (1)$$

Two factors set lower bounds on the value that the *RTO* can achieve: an explicit configuration parameter, RTO_{min}, and the implicit effects of the granularity with which RTT is measured and with which the kernel sets and checks timers. As noted earlier, common values for RTO_{min} are 200ms, and most implementations track RTTs and timers at a granularity of 1ms or larger.

Because RTT estimates are difficult to collect during loss and timeouts, a second safety mechanism controls timeout behavior—exponential backoff:

$$timeout = RTO \times 2^{backoff} \quad (2)$$

After each timeout, the retransmit timer value is doubled, helping to ensure that a single *RTO* set too low cannot cause a long-lasting chain of retransmissions. Whenever an ACK is received, the backoff parameter is reset to zero.

3.2 Evaluation Workload

Test application: Striped requests. The test client issues a request for a block of data that is striped across N servers (the "stripe width"). Each server responds with $\frac{blocksize}{N}$ bytes of data. Only after it receives the full response from every server will the client issue requests for the subsequent data block. This design mimics the request patterns found in several cluster filesystems and parallel workloads. Observe that as the number of servers increases, the amount of data requested from each server decreases. We run each experiment for 200 data block transfers to observe steady-state performance (confidence intervals are within 5%), calculating the goodput (application throughput) over the entire duration of the transfer.

Different systems have their own "natural" block sizes. We select the block size (1MB) based upon read sizes common in several distributed filesystems, such as GFS [15] and PanFS [34], which observe workloads that read on the order of a few kilobytes to a few megabytes at a time. Our prior work suggests that the block size shifts the onset of incast (doubling the block size doubles the number of servers before experiencing incast collapse), but does not substantially change the system's behavior [28]. The mechanisms we develop mitigate TCP incast collapse for any choice of block sizes and buffer sizes.

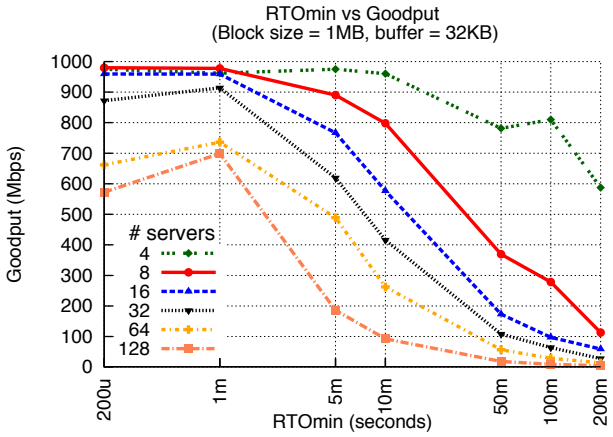


Figure 2: Reducing the RTO_{min} in simulation to microseconds from the current default value of 200ms improves goodput.

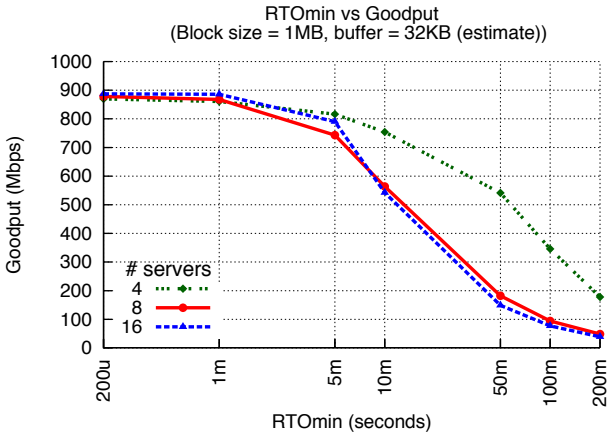


Figure 3: Experiments on a real cluster validate the simulation result that reducing the RTO_{min} to microseconds improves goodput.

3.3 In Simulation

We simulate one client and multiple servers connected through a single switch where round-trip-times under low load are 100μs. Each node has a 1Gbps capacity link, and we configure the switch buffers with 32KB of output buffer space per port, a size chosen based on statistics from commodity 1Gbps Ethernet switches. Because ns-2 is an event-based simulation, the timer granularity is infinite, hence we investigate the effect of RTO_{min} to understand how low the RTO needs to be to avoid TCP incast collapse. Additionally, we add a small random timer scheduling delay of up to 20μs to account for real-world scheduling variance.

Figure 2 plots goodput as a function of the RTO_{min} for stripe widths between 4–128 servers. Goodput, using the default 200ms RTO_{min}, drops by nearly an order of magnitude with 8 concurrent senders, and by nearly two orders of magnitude when data is striped across 64 and 128 servers.

Reducing the RTO_{min} to 1ms is effective for 8–16 concurrent senders, fully utilizing the client's link, but begins to suffer when data is striped across more servers: 128 concurrent senders utilize only 50% of the available link bandwidth even with a 1ms RTO_{min}. For 64 and 128 servers and low RTO_{min}, each flow does not have enough data to send to individually saturate the link, given the inherent inefficiency of synchronizing 64 or 128 streams each sending very little.

3.4 In Real Clusters

We study TCP incast collapse on two clusters; one sixteen-node cluster using an HP Procurve 2848 switch, and one 48-node cluster using a Force10 S50 switch. In these clusters, every node has 1 Gbps links and a client-to-server RTT of approximately 100μs. All nodes run Linux kernel 2.6.28. We run the same synchronized read workload as in simulation.

For these experiments, we modified the Linux 2.6.28 kernel to use microsecond-accurate timers with microsecond-granularity RTT estimation (Section 5) to accurately set the RTO_{min} to a desired value. Without these modifications, the TCP RTO can be reduced only to 5ms.

Figure 3 plots the application throughput as a function of the RTO_{min} for 4, 8, and 16 concurrent senders. For all configurations, goodput drops with increasing RTO_{min} above 1ms. For 8 and 16 concurrent senders, the default RTO_{min} of 200ms results in nearly 2 orders of magnitude drop in throughput.

The real world results deviate from the simulation results in a few minor ways. First, the maximum achieved throughput in simulation nears 1Gbps, whereas the maximum achieved in the real world is 900Mbps. Simulation throughput is always higher because simulated nodes are infinitely fast, whereas real-world nodes are subject to myriad delaying influences, including OS scheduling and Ethernet or switch timing differences.

Second, real world results show negligible difference between 8 and 16 servers, while the differences are more pronounced in simulation. We attribute this to variances in the buffering between simulation and the real world. Simulation statically assigns switch buffer sizes on a per-port basis, whereas many real-world switches dynamically allocate memory from a shared buffer pool. Even with dynamic allocation, however, switch buffers are often not large enough to prevent TCP incast collapse in real world cluster environments.

Third, the real world results show identical performance for RTO_{min} values of 200μs and 1ms, whereas there are slight differences in simulation. Timeouts in real-world experiments are longer than in simulation because of increased latency and RTT variance in the real-world. Figure 4 shows the distribution of round-trip-times during an incast workload in the real world. While the baseline RTTs can be between 50-100μs, increased congestion causes RTTs to rise to 400μs on average with spikes as high as 850μs. Hence, the higher RTTs combined with increased RTT variance causes the RTO estimator to set timeouts of 1–3ms, and an RTO_{min} below 1ms will not lead to shorter retransmission times. Hence, where we specify a RTO_{min} of 200μs, we are effectively eliminating the RTO_{min}, allowing RTO to be as low as calculated by equation (1).

Despite these differences, the real world results show the need to reduce the RTO to at least 1ms to avoid throughput degradation at scales of up to 16 servers. In the next section, we explain why providing microsecond-granularity retransmissions will be required for future, faster datacenter networks.

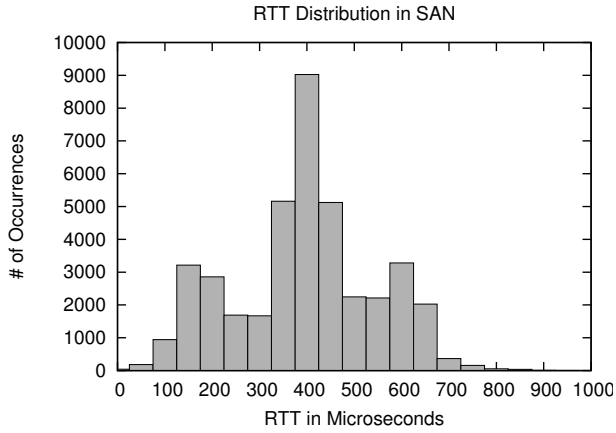


Figure 4: During an incast experiment on a cluster RTTs increase by 4 times the baseline RTT (100μs) on average with spikes as high as 800μs. This produces *RTO* values in the range of 1-3ms, resulting in an RTO_{min} of 1ms being as effective as 200μs in today's networks.

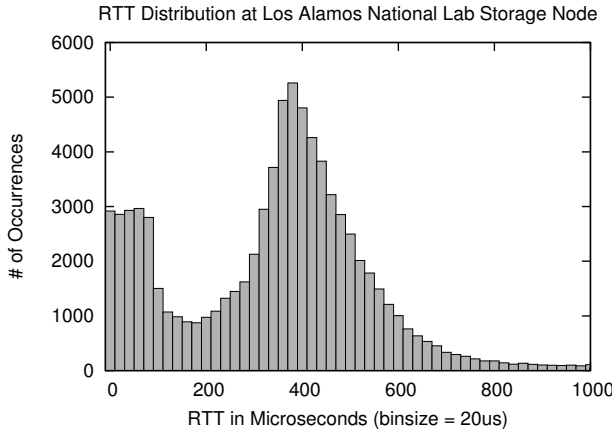


Figure 5: The distribution of RTTs from an active storage node at Los Alamos National Lab shows an appreciable number of RTTs in the 10s of microseconds.

4. NEXT-GENERATION DATACENTERS

TCP incast collapse poses more problems for the next generation of datacenters with 10Gbps networks and thousands of machines. 10Gbps networks have smaller RTTs than 1Gbps networks; port-to-port latency can be as low as 10μs. For example, we plot the distribution of RTTs from live traces of an active storage node at Los Alamos National Laboratory in Figure 5: 20% of RTTs are below 100μs even when accounting for kernel scheduling, showing that networks and systems today operate in very low-latency environments. Because 10Gbps Ethernet provides higher bandwidth, servers can send their portion of a data block more quickly, requiring smaller *RTO* values to avoid idle link time. In this section, we show the need for fine-grained TCP retransmissions for 10Gbps low latency datacenters.

4.1 Scaling to Thousands

We analyze the impact of TCP incast collapse and the reduced *RTO* solution for 10Gbps Ethernet networks in simulation as we scale the number of concurrent senders into the thousands. We reduce baseline RTTs from 100μs to 20μs and temporarily eliminate the 20μs timer scheduling variance, and increase link capacity to 10Gbps, setting per-port buffer size to 32KB based on our real-world experiments in 10Gbps cluster environments.

We increase the blocksize to 80MB to ensure each flow can individually saturate a 10Gbps link, varying the number of servers from 32 to 2048. Figure 6 shows that having an artificial bound of either 1ms or 200μs results in low goodput in a network whose RTTs are 20μs. This underscores the requirement that *retransmission timeouts should be on the same timescale as network latency* to avoid incast collapse—simply changing a constant in today's TCP implementations will not suffice.

Eliminating a lower bound on *RTO* performs well for up to 512 concurrent senders, but for 1024 servers and beyond, even the aggressively low *RTO* configuration sees up to a 50% reduction in goodput caused by significant periods of link idle time. These idle periods are caused by *repeated, simultaneous, successive timeouts*. Recall that after every timeout, the *RTO* value is doubled until an ACK is received. This has been historically safe because the exponential backoff quickly and conservatively estimates the duration to wait until congestion abates. For incast communication, however, the exponentially increasing delay can overshoot some portion of time the link is actually idle, leading to sub-optimal goodput. Because only one flow must overshoot to delay the entire transfer, the probability of overshooting increases with increased number of flows.

Figure 7 shows a client's instantaneous link utilization and the retransmission events for one of the flows that experienced repeated retransmission losses during an incast simulation on a 1Gbps network. This flow timed out and retransmitted a packet at the same time that other timed out flows also retransmitted. While some of these flows got through and saturated the link for a brief period of time, the flow shown here timed out and doubled its timeout value (until the maximum factor of 64 * *RTO*) following each failed retransmission. Often the link became available shortly after the retransmission event, but the retransmission timer was set to fire far beyond this time. When a retransmission was successful, the block transfer completed and the next block transfer began, but only after large periods of link idle time that reduced goodput.

In summary, decreased goodput for a large number of flows can be attributed to many flows timing out simultaneously, backing off *deterministically*, and retransmitting at precisely the same time. While some of the flows are successful on this retransmission, a majority of flows lose their retransmitted packet and backoff by another factor of two, sometimes far beyond when the link becomes idle.

4.2 Desynchronizing Retransmissions

By adding some randomness to the *RTO*, the retransmissions can be desynchronized so that fewer flows experience repeated timeouts when RTO_{min} is removed. We examine the retransmission synchronization effect in simulation, measuring the goodput for several different settings. In Figure 8,

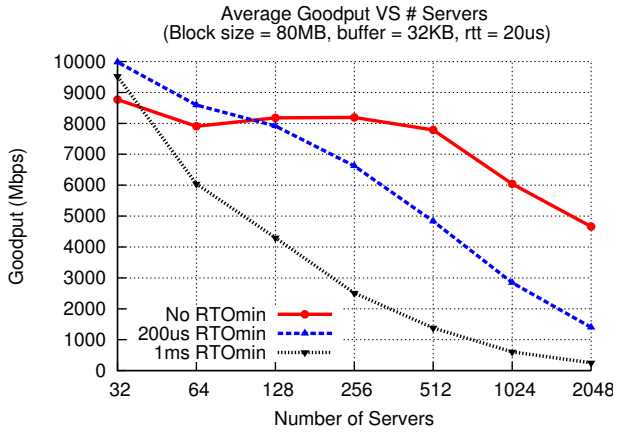


Figure 6: In simulation, flows experience reduced goodput when retransmissions do not fire at the same granularity as RTTs. Fine-grained timers can observe suboptimal goodput for a large number of servers if retransmissions are tightly synchronized.

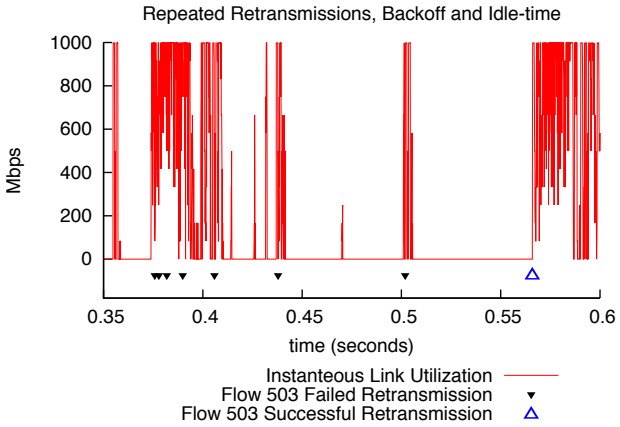


Figure 7: Some flows experience repeated retransmission failures due to synchronized retransmission behavior, delaying transmission far beyond when the link is idle.

we vary the inaccuracy of scheduling to better understand how synchronized the retransmissions must be to observe repeated retransmission timeouts. When retransmissions are sent precisely when the retransmission timer fires, goodput drops significantly for a large number of concurrent senders. Adding up to 5μs of random delay helps to desynchronize some of the retransmissions in simulation. But perhaps this is a simulation artifact as real world scheduling is not without variance.

To better understand scheduling variance for retransmissions on real systems, we measured the accuracy of executing `usleep(50)` calls that use the high-precision timer subsystem we use in Section 5, finding that the sleep durations were clustered within 2–3μs, suggesting that real-world scheduling may be accurate enough to require desynchronizing retransmissions. Should TCP offload be enabled for faster, low-latency 10Gbps Ethernet cards, packet scheduling variance might be even lower. Given that round-trip-times in Figure 5 can be under 20μs, today's systems are capable of such accurate packet scheduling.

Figure 8 shows that explicitly adding an adaptive randomized *RTO* component to the scheduled timeout as follows:

$$timeout = (RTO + (rand(0.5) \times RTO)) \times 2^{backoff} \quad (3)$$

performs well regardless of the number of concurrent senders because it explicitly desynchronizes the retransmissions of flows that experience repeated timeouts, and does not heavily penalize flows that experience a few timeouts.

While we advocate an *RTO* calculation that adds a randomized component for datacenters, we have not evaluated its impact for wide-area flows, where adding a delay of up to 50% of the calculated *RTO* will increase latency—synchronized retransmissions are less likely to occur in the wide-area because flows have different RTTs and hence varying *RTO*s. Also, despite our efforts to add scheduling variance in the simulation experiments, real-world variances may be large enough to avoid having to explicitly randomize the *RTO* in practice. However, with a large number of concurrent

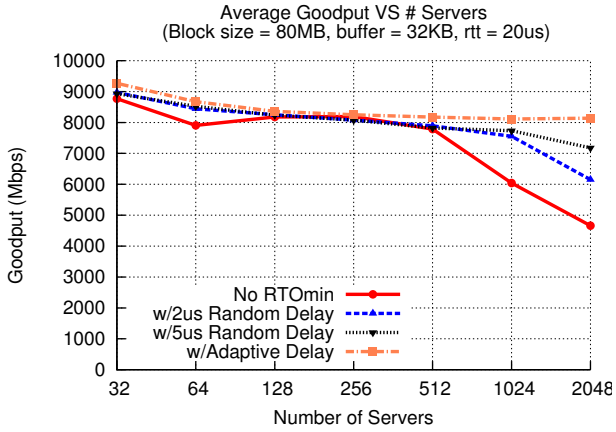


Figure 8: In simulation, introducing a randomized component to the *RTO* desynchronizes retransmissions following timeouts and avoids goodput degradation for a large number of flows.

senders, the number of flows that retransmit within a fixed period of congestion on retransmission will eventually be high enough that the repeated retransmission loss behavior shown in simulation may occur in the real-world.

In summary, we emphasize that for future low-latency datacenter networks, extremely fine-grained TCP retransmissions must be provided to avoid TCP incast collapse. Next, we discuss our implementation and evaluation of microsecond granularity TCP timeouts in the Linux operating system.

5. IMPLEMENTING FINE-GRAINED RETRANSMISSIONS

TCP implementations typically use a coarse-grained periodic timer that provides timeout support with very low overhead. Providing tighter TCP timeouts requires not only reducing or eliminating RTO_{min}, but also supporting fine-grained RTT measurements and kernel timers.

The TCP clock granularity in most popular operating

systems is on the order of milliseconds, as defined by a global counter updated by the kernel at a frequency "HZ", where HZ is typically 100, 250, or 1000. Linux, for example, updates its "jiffy" timer 250 times per second, yielding a TCP clock granularity of 4ms, with a configuration option to update 1000 times per second for a 1ms granularity. More frequent updates, as would be needed to achieve finer granularity timeouts, would impose a system-wide clock maintenance overhead considered unacceptable by most.

Unfortunately, setting the RTO_{min} to 1 jiffy (the lowest possible value) does not achieve *RTO* values of 1ms because of the clock granularity. TCP measures RTTs in 1ms granularity at best, so both the smoothed RTT estimate and RTT variance have a 1 jiffy (1ms) lower bound. Since the standard *RTO* estimator sums the RTT estimate with 4x the RTT variance, the lowest possible *RTO* value is 5 jiffies. We experimentally validated this result by setting the clock granularity to 1ms, setting RTO_{min} to 1ms, and observing that TCP timeouts were a minimum of 5ms.

At a minimum possible RTO_{min} of 5ms in standard TCP implementations, Figures 9 and 10 show that throughput collapse is significantly improved for small numbers of servers. This simple change is a good first step for use in the field today. However, application throughput is reduced by 35% or more with just 16 concurrent senders. Next, we describe how to achieve microsecond granularity *RTO* values in the real world.

5.1 Linux high-resolution timers: hrtimers

High resolution timers were introduced into Linux kernel version 2.6.18 and are still under active development [16]. They form the basis of the posix-timer and itimer user-level timers, nanosleep, and a few other in-kernel operations, including the update of the jiffies value.

The Generic Time of Day (GTOD) framework provides the kernel and other applications with nanosecond resolution timekeeping using the CPU cycle counter on all modern processors—our modifications use this framework to provide fine-grained measurements of round-trip-times. The hrtimer implementation interfaces with the High Precision Event Timer (HPET) hardware also available on modern chipsets to achieve microsecond resolution event notification in the kernel. Specifically, the HPET is a programmable hardware timer that consists of a free-running upcounter and several comparators and registers, which modern operating systems can set. When scheduling an event, the kernel sets a register value to achieve a desired interrupt interval, and the comparator will signal a hardware interrupt when the upcounter matches the register value. For example, the kernel may request the HPET to interrupt once every 1ms to update the jiffy counter, or it may set a timer for 50μs in the future for a `usleep(50)` system call.

Implementing TCP timeouts using the hrtimer subsystem could lead to increased interrupt overhead only if timeouts are frequent; flows that experience few losses do not incur hrtimer-based interrupts because the retransmission timer and HPET entry are reset for every ACK received. Our preliminary evaluations using our implementation of TCP timers using the hrtimer system have shown no noticeable overhead: while serving as a sender in an incast workload, the speed of a kernel build was about the same for both hrtimer and normal TCP implementations. We also argue that during incast workloads, a small overhead may be acceptable, as it removes the idle periods that prevent the server from doing useful work to begin with, but we leave an extensive evaluation of overhead for future work.

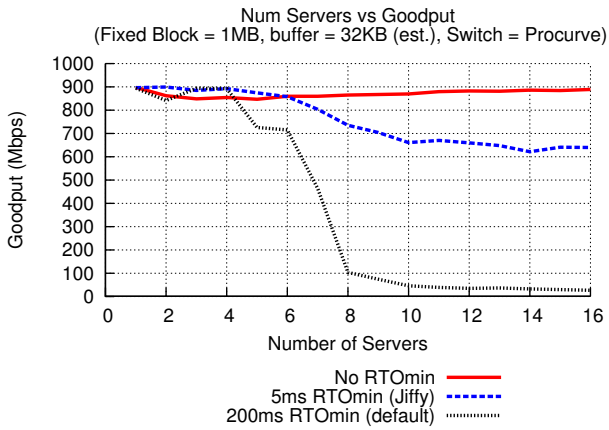


Figure 9: On a 16 node cluster, our high-resolution TCP timer modifications help eliminate incast collapse. The jiffy-based implementation has a 5ms lower bound on *RTO*, and achieves only 65% throughput.

5.2 Modifications to the TCP Stack

The Linux TCP implementation requires three changes to support microsecond timeouts using hrtimers: microsecond resolution time accounting to track RTTs with greater precision, redefinition of TCP constants, and replacement of low-resolution timers with hrtimers.

By default, the jiffy counter is used for tracking time. To provide microsecond granularity accounting, we use the GTOD framework to access the 64-bit nanosecond resolution hardware clock wherever the jiffies time is traditionally used.

With the TCP timestamp option enabled, RTT estimates are calculated based on the difference between the timestamp option in an earlier packet and the corresponding ACK. We convert the time from nanoseconds to microseconds and store the value in the TCP timestamp option.[2] This change can be accomplished entirely on the sender—receivers already echo back the value in the TCP timestamp option.

All timer constants previously defined with respect to the jiffy timer are converted to absolute values (e.g., 1ms instead of 1 jiffy). Last, the TCP implementation must make use of the hrtimer interface: we replace the standard timer objects in the socket structure with the hrtimer structure, ensuring that all subsequent calls to set, reset, or clear these timers use the appropriate hrtimer functions.

We note that the changes required to TCP were relatively minimal and non-invasive. The successful implementation of fine-grained retransmissions in TCP took two weeks for a graduate student with little experience in kernel hacking or prior exposure to Linux TCP source code. We are also making a patch available for testing.[3]

[2]The lower wrap-around time – 2^{32} microseconds or 4294 seconds – is still far greater than the maximum IP segment lifetime (120-255 seconds)

[3]See `http://www.pdl.cmu.edu/Incast/` for details

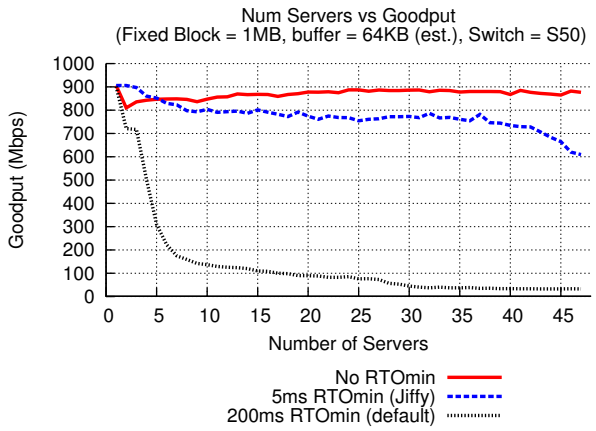


Figure 10: For a 48-node cluster, providing TCP retransmissions in microseconds eliminates incast collapse for up to 47 servers.

5.3 hrtimer Results

Figure 9 presents the achieved goodput as we increase the number of servers N using various RTO_{min} values on a Procurve 2848 switch. As before, the client issues requests for 1MB data blocks striped over N servers, issuing the next request once the previous data block has been received. Using the default 200ms RTO_{min}, throughput plummets beyond 8 concurrent senders. For a 5ms jiffy-based RTO_{min}, throughput begins to drop at 8 servers to about 70% of link capacity and slowly decreases thereafter. Last, our TCP hrtimer implementation allowing microsecond *RTO* values achieves the maximum achievable goodput for 16 concurrent senders.

We verify these results on a second cluster consisting of 1 client and 47 servers connected to a single 48-port Force10 S50 switch (Figure 10). The microsecond *RTO* kernel is again able to saturate throughput up to 47 servers. The 5ms RTO_{min} jiffy-based configuration obtained 70-80% throughput, with an observable drop above 40 concurrent senders.

Overall, we find that enabling microsecond *RTO* values in TCP successfully avoids TCP incast collapse in two real-world clusters for as many as 47 concurrent servers, and that microsecond resolution is *necessary* to achieve full performance.

6. IMPLICATIONS OF FINE-GRAINED TCP RETRANSMISSIONS

Eliminating RTO_{min} and enabling TCP retransmissions in microseconds helps avoid TCP incast collapse. But proposing fine-grained retransmissions requires addressing the issue of safety and generality: is an aggressive timeout appropriate for use in general (i.e., in the wide area), or should it be limited to the datacenter? Does it risk increased congestion or decreased throughput because of spurious (incorrect) timeouts? In this section, we discuss the implications of this change on wide-area bulk transfers and on delayed acknowledgments.

6.1 Is it safe to eliminate RTOmin?

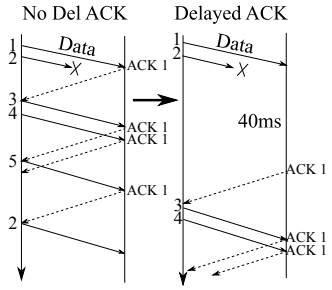
There are two possible complications of permitting much smaller *RTO* values: spurious (incorrect) timeouts when the network RTT suddenly jumps, and breaking the relationship between the delayed acknowledgement timer and the *RTO* values.

Spurious retransmissions: The most prominent study of TCP retransmission by Allman and Paxson showed that a high (by the standards of datacenter RTTs) RTO_{min} helped avoid spurious retransmission in wide-area TCP transfers [2], regardless of how good an estimator one used based on historical RTT information. Intuition for why this is the case comes from prior [24, 10] and subsequent [35] studies of Internet delay changes. While most of the time, end-to-end delay can be modeled as random samples from some distribution (and therefore, can be predicted by the *RTO* estimator in equation (1)), end-to-end delay consistently observes both occasional, unpredictable delay spikes, as well as shifts in the distribution. Such changes can be due to the sudden introduction of cross-traffic, routing changes, or failures. As a result, wide-area "packet delays [are] not mathematically [or] operationally steady" [35], which confirms the Allman and Paxson observation that *RTO* estimation involves a fundamental tradeoff between rapid retransmission and spurious retransmissions.

Delayed Acknowledgements: The TCP delayed ACK mechanism attempts to reduce the amount of ACK traffic by having a receiver acknowledge only every other packet [7]. If a single packet is received with none following, the receiver will wait up to the delayed ACK timeout threshold before sending an ACK.

Prior work showed that in cluster based systems with three to five servers and with barrier-synchronized request workloads, the delayed ACK mechanism can act as a miniature timeout, resulting in reduced, but not catastrophically low, throughput during certain loss patterns [28].

The figure on the right shows how the combination of small windows and delayed ACKs can result in lower throughput and slower loss recovery. With delayed ACK disabled, the ACK for packet 1 is sent immediately after it is received, enabling the sender to grow its TCP window and triggering data-driven recovery for packet 2. With delayed ACKs enabled, the first ACK is sent 40ms later, delaying this recovery process.



While this delay is not as high as a full 200ms *RTO*, the default delayed ACK minimum (40ms in Linux) is still large compared to the RTTs in datacenters and results in low throughput for three to five concurrent senders. Beyond five senders, high packet loss results in 200ms retransmission timeouts which mask the impact of delayed ACK-induced link idle time.

Microsecond retransmission timeouts, however, have a different interaction with the delayed ACK mechanism. The receiver's delayed ACK timer should always fire before the sender's retransmission timer fires to prevent the sender from timing out waiting for an ACK that is merely delayed. Modern systems protect against this by setting the delayed ACK timer to a value (40ms) that is safely under the RTO_{min} (200ms).

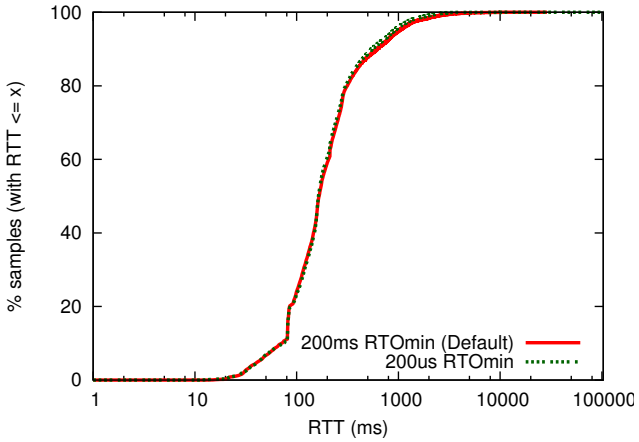


Figure 11: A comparison of RTT distributions of flows collected over 3 days on the two configurations show that both servers saw a similar distribution of both short and long-RTT flows.

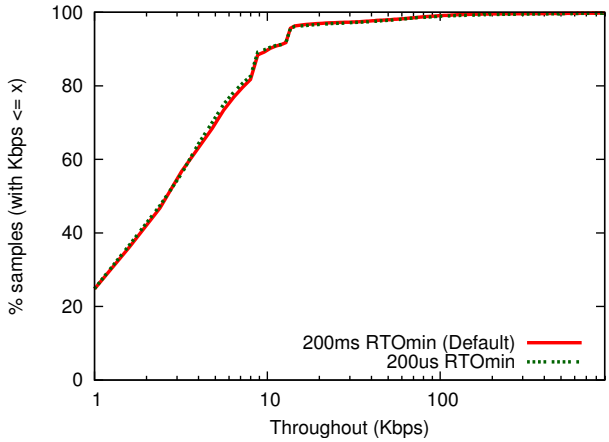
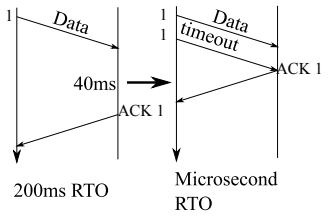


Figure 12: The two configurations observed an identical throughput distribution for flows. Only flows with throughput over 100 bits/s were considered.

As depicted to the right, a host with microsecond-granularity retransmissions would periodically experience an unnecessary timeout when communicating with unmodified hosts where the *RTO* is below 40ms (e.g., in the datacenter and for short flows in the wide-area), because the sender incorrectly assumes that a loss has occurred.



Given these consequences, there are good reasons to ask whether eliminating RTO_{min}—basing timeouts solely upon the Jacobson estimator and exponential backoff—will harm wide-area performance and datacenter environments with clients using delayed ACK. In practice, these two potential consequences are mitigated by newer TCP features and by the limited circumstances in which they occur, as we explore in the next two sections. We find that eliminating the RTO_{min} has little impact on bulk data transfer performance for wide-area flows, and that in the datacenter, delayed ACK causes only a small, though noticeable drop in throughput when the RTO_{min} is set below the delayed ACK threshold.

6.2 In the Wide Area

Aggressively lowering both the *RTO* and RTO_{min} shows practical benefits for datacenters. In this section, we investigate if reducing the RTO_{min} value to microseconds and using finer granularity timers is safe for wide area transfers. We find that the impact of spurious timeouts on long, bulk data flows is very low – within the margins of error – allowing *RTO* to go into the microseconds without impairing wide-area performance.

The major potential effect of a spurious timeout is a loss of performance: a flow that experiences a timeout will reduce its slow-start threshold (ssthresh) by half, its window to one and attempt to rediscover link capacity. It is important to understand that spurious timeouts do not endanger network stability through increased congestion [2]. Spurious timeouts occur not when the network path drops packets, but rather when the path observes a sudden, higher delay. Because a TCP sender backs-off on the amount of data it injects into the network following this timeout, the effect of a shorter *RTO* on increased congestion is likely small. Therefore, we analyze the *performance* of TCP flows over the wide-area for bulk data transfers.

Fortunately, algorithms to undo the effects of spurious timeouts have been both proposed [2, 21, 32] and, in the case of F-RTO [32], adopted in the latest Linux implementations. The default F-RTO settings conservatively halve the congestion window when a spurious timeout is detected but remain in congestion avoidance mode, thus avoiding the slow-start phase. Therefore, the impact of spurious timeouts on throughput are now significantly smaller than they were 10 years ago.

6.2.1 Experimental Setup and Results

We deployed two servers that differ only in their implementation of the *RTO* values and granularity, one using the default Linux 2.6.28 kernel with a 200ms RTO_{min}, and the other using our modified hrtimer-enabled TCP stack with a 200μs RTO_{min}. We downloaded 12 torrent files consisting of various Linux distributions and began seeding all content from both machines on the same popular swarms for three days. Each server uploaded over 30GB of data, and observed around 70,000 flows (with non-zero throughput) over the course of three days. We ran tcpdump on each machine to collect all uploaded traffic packet headers for later analysis.

The TCP *RTO* value is determined by the estimated RTT value of each flow. Other factors being equal, TCP throughput tends to decrease with increased RTT. To compare *RTO* and throughput for the 2 servers, we first investigate if they see similar flows with respect to RTT values. Figure 11 shows the per-flow average RTT distribution for both hosts over the three day measurement period. The RTT distributions are nearly identical, suggesting that each machine saw a similar distribution of both short- and long-RTT flows. The per-packet RTT distribution for both flows is also identical.

Figure 12 shows the per-flow throughput distributions for both hosts, filtering out flows with throughput less than

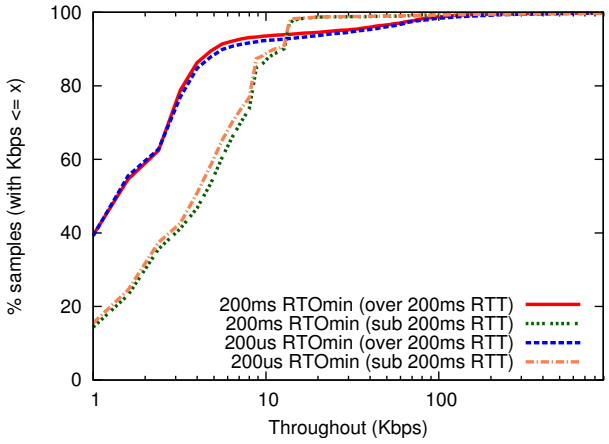


Figure 13: The throughput distribution for short and long RTT flows shows negligible difference across configurations.

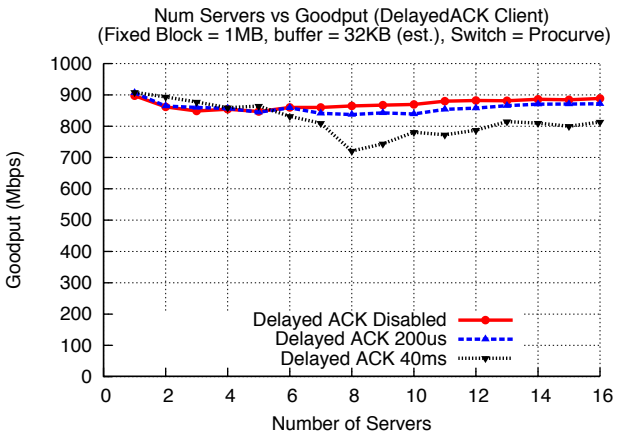


Figure 14: With RTO_{min} eliminated, disabling delayed ACK on client nodes provides optimal goodput in a 16-node cluster.

100bps, which are typically flows sending small control packets. The throughput distributions are also nearly identical—the host with $RTO_{min} = 200\mu$s did not perform worse on the whole than the host with $RTO_{min} = 200$ms.

We split the throughput distributions based on whether the flow's RTT was above or below 200ms. For flows above 200ms, we use the variance in the two distributions as a control parameter: any variance seen above 200ms are a result of measurement noise, because the RTO_{min} is no longer a factor. Figure 13 shows that the difference between the distribution for flows below 200ms is within this measurement noise.

This data suggests that reducing the RTO_{min} to 200μs in practice does not affect the performance of bulk-data TCP flows on the wide-area.

6.3 Interaction with Delayed ACK

For servers using a reduced *RTO* in a datacenter environment, the server's retransmission timer may expire long before an unmodified client's 40ms delayed ACK timer fires. As a result, the server will timeout and resend the unacked packet, cutting ssthresh in half and rediscovering link capacity using slow-start. Because the client acknowledges the retransmitted segment immediately, the server does not observe a coarse-grained 40ms delay, only an unnecessary timeout.

Once the first-order throughput collapse has been averted by having the sender use a microsecond-granularity *RTO*, Figure 14 shows the performance difference between a client with delayed ACK disabled, delayed ACK enabled with a 200μs timer, and the default 40ms delayed ACK configuration.

Beyond 8 servers, a client with a 200μs delayed ACK timer receives 15–30Mbps lower throughput compared to a client with delayed ACK disabled entirely, whereas the 40ms delayed ACK client experiences between 100 and 200Mbps lower throughput caused by frequent timeouts. The 200μs delayed ACK timeout client delays a server by roughly a round-trip-time and does not force the server to timeout, so the performance hit is much smaller.

Delayed ACK can provide benefits where the ACK path is congested [4], but in the datacenter environment, we believe that coarse-grained delayed ACKs should be avoided when possible; most high-performance applications in the datacenter favor quick response over additional ACK-processing overhead and are typically equally provisioned for both directions of traffic. Our evaluations in Section 5 disable delayed ACK on the client for this reason. While these results show that for full performance, delayed ACK should be disabled, we note that unmodified clients still achieve good performance and avoid incast collapse when the servers only implement fine-grained retransmissions.

7. RELATED WORK

TCP Improvements: A number of changes over the years have improved TCP's ability to respond to loss patterns and perform better in particular environments, many of which are relevant to the high-performance datacenter environment we study. NewReno and SACK, for instance, reduce the number of loss patterns that will cause timeouts; prior work on the TCP incast problem showed that NewReno, in particular, improved throughput during moderate amounts of incast traffic, though not when the problem became severe [28].

TCP mechanisms such as Limited Transmit [1] were specifically designed to help TCP recover from packet loss when window sizes are small—exactly the problem that occurs during incast collapse. This solution again helps maintain throughput under modest congestion, but during severe incast collapse, the most common loss pattern is the loss of the entire window.

Finally, proposed improvements to TCP such as TCP Vegas [8] and FAST TCP [19] can limit window growth when RTTs begin to increase, often combined with more aggressive window growth algorithms to rapidly fill high bandwidth-delay links. Unlike the self-interfering oscillatory behavior on high-BDP links that this prior work seeks to resolve, incast collapse is triggered by the arrival and rapid ramp-up of numerous competing flows, and the RTT increases drastically (or becomes a full window loss) over a single round-trip. While an RTT-based approach is an interesting approach to study for alternative solutions to incast collapse, it is a matter of considerable future work to adapt existing

techniques for this purpose.

Efficient, fine-grained kernel timers. Where our work depends on hardware support for high-resolution kernel timers, earlier work on "soft timers" shows an implementation path for legacy systems [3]. Soft timers can provide microsecond-resolution timers for networking without introducing the overhead of context switches and interrupts. The hrtimer implementation we make use of draws lessons from soft timers, using a hardware interrupt to trigger all available software interrupts.

Understanding RTO_{min}. The origin of concern about the safety and generality of reducing RTO_{min} was presented by Allman and Paxson [2], where they used trace-based analysis to show that there existed no optimal *RTO* estimator, and to what degree that the TCP granularity and RTO_{min} had an impact on spurious retransmissions. Their analysis showed that a low or non-existent RTO_{min} greatly increased the chance of spurious retransmissions and that tweaking the RTO_{min} had no obvious sweet-spot for balancing fast response with spurious timeouts. They showed the increased benefit of having a fine measurement granularity for responding to good timeouts because of the ability to respond to minor changes in RTT. Last, they suggested that the impact of bad timeouts could be mitigated by using the TCP timestamp option, which later became known as the Eifel algorithm [21]. F-RTO later showed how to detect spurious timeouts by detecting whether the following acknowledgements were for segments not retransmitted [32], and this algorithm is implemented in Linux TCP today.

Psaras and Tsaoussidis revisit the minimum *RTO* for high-speed, last-mile wireless links, noting the default RTO_{min} is responsible for worse throughput on wireless links and short flows [29]. They suggest a mechanism for dealing with delayed ACKs that attempts to predict when a packet's ACK is delayed—a per-packet RTO_{min}. We find that while delayed ACK can affect performance for low RTO_{min}, the benefits of a low RTO_{min} far outweigh the impact of delayed ACK on performance.

Concurrent work is studying the possible effects of TCP incast collapse in other datacenter workloads [9], such as in MapReduce [11], independently confirming that faster TCP retransmissions can help improve goodput for these alternative workloads.

8. CONCLUSION

This paper presented a *practical*, *effective*, and *safe* solution to eliminate TCP incast collapse in datacenter environments. Enabling microsecond-granularity TCP timeouts allowed high-fan-in, barrier synchronized datacenter communication to scale to 47 nodes in a real cluster evaluation, and randomized retransmissions were used to scale to thousands of nodes in simulation. This implementation of fine-grained TCP retransmissions should also help latency-sensitive datacenter applications where timeouts lasting hundreds of milliseconds can harm response time. Through a wide-area evaluation, we showed that these modifications remain safe for use in the wide-area, providing a general and effective improvement for TCP-based cluster communication.

Acknowledgments

We would like to thank our shepherd Dave Maltz, Dilip Chhetri, Vyas Sekar, Srinivasan Seshan, and the anonymous reviewers for their comments and suggestions. We also thank Andrew Shewmaker, HB Chen, Parks Fields, Gary Grider, Ben McClelland, and James Nunez at Los Alamos National Laboratory for help with obtaining packet header traces.

We thank the members and companies of the PDL Consortium (including APC, DataDomain, EMC, Facebook, Google, Hewlett-Packard, Hitachi, IBM, Intel, LSI, Microsoft, NEC, NetApp, Oracle, Seagate, Sun, Symantec, and VMware) for their interest, insights, feedback, and support. We thank Intel and NetApp for hardware donations that enabled this work.

This material is based upon research supported in part by the National Science Foundation via grants CNS-0546551, CNS-0619525, CNS-0326453, and CCF-0621499, by the Army Research Office, under agreement number DAAD19-02-1-0389, by the Department of Energy, under Award Number DE-FC02-06ER25767, and by Los Alamos National Laboratory, under contract number 54515-001-07.

9. REFERENCES

[1] M. Allman, H. Balakrishnan, and S. Floyd. *Enhancing TCP's Loss Recovery Using Limited Transmit*. Internet Engineering Task Force, Jan. 2001. RFC 3042.

[2] M. Allman and V. Paxson. On estimating end-to-end network path properties. In *Proc. ACM SIGCOMM*, Cambridge, MA, Sept. 1999.

[3] M. Aron and P. Druschel. Soft timers: Efficient microsecond software timer support for network processing. *ACM Transactions on Computer Systems*, 18(3):197–228, 2000.

[4] H. Balakrishnan, V. N. Padmanabhan, and R. Katz. The effects of asymmetry on TCP performance. In *Proc. ACM MOBICOM*, Budapest, Hungary, Sept. 1997.

[5] H. Balakrishnan, V. N. Padmanabhan, S. Seshan, and R. Katz. A comparison of mechanisms for improving TCP performance over wireless links. In *Proc. ACM SIGCOMM*, Stanford, CA, Aug. 1996.

[6] P. J. Braam. File systems for clusters from a protocol perspective. `http://www.lustre.org`.

[7] R. T. Braden. *Requirements for Internet Hosts—Communication Layers*. Internet Engineering Task Force, Oct. 1989. RFC 1122.

[8] L. S. Brakmo, S. W. O'Malley, and L. L. Peterson. TCP vegas: New techniques for congestion detection and avoidance. In *Proc. ACM SIGCOMM*, London, England, Aug. 1994.

[9] Y. Chen, R. Griffith, J. Liu, A. D. Joseph, and R. H. Katz. Understanding TCP incast throughput collapse in datacenter networks. In *Proc. Workshop: Research on Enterprise Networking*, Barcelona, Spain, Aug. 2009.

[10] k. claffy, G. Polyzos, and H.-W. Braun. Measurement considerations for assessing unidirectional latencies. *Internetworking: Research and Experience*, 3(4):121–132, Sept. 1993.

[11] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proc. 6th USENIX OSDI*, San Francisco, CA, Dec. 2004.

[12] Scaling memcached at Facebook. `http://www.facebook.com/note.php?note_id=39391378919`.

[13] S. Floyd and V. Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking*, 1(4), Aug. 1993.

[14] B. Ford. Structured streams: A new transport abstraction. In *Proc. ACM SIGCOMM*, Kyoto, Japan, Aug. 2007.

[15] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *Proc. 19th ACM Symposium on Operating Systems Principles (SOSP)*, Lake George, NY, Oct. 2003.

[16] High-resolution timer subsystem. `http://www.tglx.de/hrtimers.html`.

[17] V. Jacobson. Congestion avoidance and control. In *Proc. ACM SIGCOMM*, pages 314–329, Vancouver, British Columbia, Canada, Sept. 1998.

[18] V. Jacobson, R. Braden, and D. Borman. *TCP Extensions for High Performance*. Internet Engineering Task Force, May 1992. RFC 1323.

[19] C. Jin, D. X. Wei, and S. H. Low. FAST TCP: motivation, architecture, algorithms, performance.

[20] E. Kohler, M. Handley, and S. Floyd. Designing DCCP: Congestion control without reliability. In *Proc. ACM SIGCOMM*, Pisa, Italy, Aug. 2006.

[21] R. Ludwig and M. Meyer. *The Eifel Detection Algorithm for TCP*. Internet Engineering Task Force, Apr. 2003. RFC 3522.

[22] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. *TCP Selective Acknowledgment Options*. Internet Engineering Task Force, 1996. RFC 2018.

[23] A distributed memory object caching system. `http://www.danga.com/memcached/`.

[24] A. Mukherjee. On the dynamics and significance of low frequency components of Internet load. *Internetworking: Research and Experience*, 5:163–205, Dec. 1994.

[25] D. Nagle, D. Serenyi, and A. Matthews. The Panasas ActiveScale Storage Cluster: Delivering scalable high bandwidth storage. In *SC '04: Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*, Washington, DC, USA, 2004.

[26] ns-2 Network Simulator. `http://www.isi.edu/nsnam/ns/`, 2000.

[27] C. Partridge. *Gigabit Networking*. Addison-Wesley, Reading, MA, 1994.

[28] 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 *Proc. USENIX Conference on File and Storage Technologies*, San Jose, CA, Feb. 2008.

[29] I. Psaras and V. Tsaoussidis. The TCP minimum RTO revisited. In *IFIP Networking*, May 2007.

[30] K. Ramakrishnan and S. Floyd. *A Proposal to Add Explicit Congestion Notification (ECN) to IP*. Internet Engineering Task Force, Jan. 1999. RFC 2481.

[31] S. Raman, H. Balakrishnan, and M. Srinivasan. An image transport protocol for the Internet. In *Proc. International Conference on Network Protocols*, Osaka, Japan, Nov. 2000.

[32] P. Sarolahti and M. Kojo. *Forward RTO-Recovery (F-RTO): An Algorithm for Detecting Spurious Retransmission Timeouts with TCP and the Stream Control Transmission Protocol (SCTP)*. Internet Engineering Task Force, Aug. 2005. RFC 4138.

[33] S. Shepler, M. Eisler, and D. Noveck. NFSv4 Minor Version 1 – Draft Standard. `http://www.ietf.org/internet-drafts/draft-ietf-nfsv4-minorversion1-29.txt`.

[34] B. Welch, M. Unangst, Z. Abbasi, G. Gibson, B. Mueller, J. Zelenka, and B. Zhou. Scalable performance of the Panasas parallel file system. In *Proc. USENIX Conference on File and Storage Technologies*, San Jose, CA, Feb. 2008.

[35] Y. Zhang, N. Duffield, V. Paxson, and S. Shenker. On the constancy of Internet path properties. In *Proc. ACM SIGCOMM Internet Measurement Workshop*, San Fransisco, CA, Nov. 2001.