TIMELY: RTT-based Congestion Control for the Datacenter

Radhika Mittal*(UC Berkeley), Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi*(Microsoft), Amin Vahdat, Yaogong Wang, David Wetherall, David Zats

Google, Inc.

ABSTRACT

Datacenter transports aim to deliver low latency messaging together with high throughput. We show that simple packet delay, measured as round-trip times at hosts, is an effective congestion signal without the need for switch feedback. First, we show that advances in NIC hardware have made RTT measurement possible with microsecond accuracy, and that these RTTs are sufficient to estimate switch queueing. Then we describe how TIMELY can adjust transmission rates using RTT gradients to keep packet latency low while delivering high bandwidth. We implement our design in host software running over NICs with OS-bypass capabilities. We show using experiments with up to hundreds of machines on a Clos network topology that it provides excellent performance: turning on TIMELY for OS-bypass messaging over a fabric with PFC lowers 99 percentile tail latency by 9X while maintaining near line-rate throughput. Our system also outperforms DCTCP running in an optimized kernel, reducing tail latency by 13X. To the best of our knowledge, TIMELY is the first delay-based congestion control protocol for use in the datacenter, and it achieves its results despite having an order of magnitude fewer RTT signals (due to NIC offload) than earlier delay-based schemes such as Vegas.

CCS Concepts

•**Networks** → **Transport protocols;**

Keywords

datacenter transport; delay-based congestion control; OS-bypass; RDMA

*Work done while at Google

SIGCOMM '15 August 17-21, 2015, London, United Kingdom

ACM ISBN 978-1-4503-3542-3/15/08.

DOI: http://dx.doi.org/10.1145/2785956.2787510

1. INTRODUCTION

Datacenter networks run tightly-coupled computing tasks that must be responsive to users, e.g., thousands of back-end computers may exchange information to serve a user request, and all of the transfers must complete quickly enough to let the complete response to be satisfied within 100 ms [24]. To meet these requirements, datacenter transports must simultaneously deliver high bandwidth (≫Gbps) and utilization at low latency (≪msec), even though these aspects of performance are at odds. Consistently low latency matters because even a small fraction of late operations can cause a ripple effect that degrades application performance [21]. As a result, datacenter transports must strictly bound latency and packet loss.

Since traditional loss-based transports do not meet these strict requirements, new datacenter transports [10, 18, 30, 35, 37, 47], take advantage of network support to signal the onset of congestion (e.g., DCTCP [35] and its successors use ECN), introduce flow abstractions to minimize completion latency, cede scheduling to a central controller, and more. However, in this work we take a step back in search of a simpler, immediately deployable design.

The crux of our search is the congestion signal. An ideal signal would have several properties. It would be fine-grained and timely to quickly inform senders about the extent of congestion. It would be discriminative enough to work in complex environments with multiple traffic classes. And, it would be easy to deploy.

Surprisingly, we find that a well-known signal, properly adapted, can meet all of our goals: delay in the form of RTT measurements. RTT is a fine-grained measure of congestion that comes with every acknowledgment. It effectively supports multiple traffic classes by providing an inflated measure for lower-priority transfers that wait behind higher-priority ones. Further, it requires no support from network switches.

Delay has been explored in the wide-area Internet since at least TCP Vegas [16], and some modern TCP variants use delay estimates [44, 46]. But this use of delay has not been without problems. Delay-based schemes tend to compete poorly with more aggressive, loss-based schemes, and delay

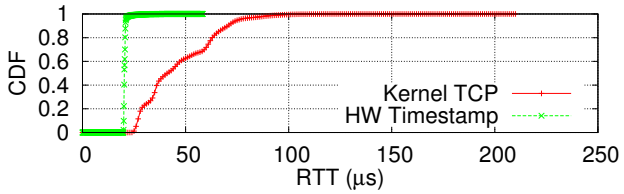

Figure 1: RTTs measured by hardware timestamps have a much smaller random variance than that by kernel TCP stack.

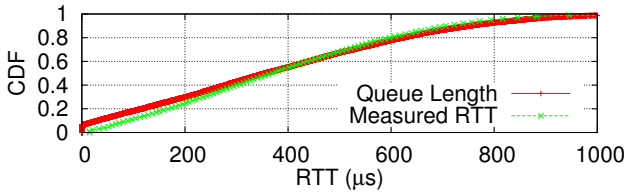

Figure 2: RTTs measured at end-system track closely the queue occupancy at congested link.

estimates may be wildly inaccurate due to host and network issues, e.g., delayed ACKs and different paths. For these reasons, delay is typically used in hybrid schemes with other indicators such as loss.

Moreover, delay has not been used as a congestion signal in the datacenter because datacenter RTTs are difficult to measure at microsecond granularity. This level of precision is easily overwhelmed by host delays such as interrupt processing for acknowledgments. DCTCP eschews a delay-based scheme saying "the accurate measurement of such small increases in queueing delay is a daunting task." [35]

Our insight is that recent NIC advances do allow datacenter RTTs to be measured with sufficient precision, while the wide-area pitfalls of using delay as a congestion signal do not apply. Recent NICs provide hardware support for high-quality timestamping of packet events [1,3,5,8,9], plus hardware-generated ACKs that remove unpredictable host response delays. Meanwhile, datacenter host software can be controlled to avoid competition with other transports, and multiple paths have similar, small propagation delays.

In this paper, we show that delay-based congestion control provides excellent performance in the datacenter. Our key contributions include:

1. We experimentally demonstrate how multi-bit RTT signals measured with NIC hardware are strongly correlated with network queueing.
2. We present *Transport Informed by MEasurement of LatencY* (TIMELY): an RTT-based congestion control scheme. TIMELY uses rate control and is designed to work with NIC offload of multi-packet segments for high performance. Unlike earlier schemes [16,46], we do not build the queue to a fixed RTT threshold. Instead, we use the rate of RTT variation, or the *gradient*, to predict the onset of congestion and hence keep the delays low while delivering high throughput.
3. We evaluate TIMELY with an OS-bypass messaging implementation using hundreds of machines on a Clos network topology. Turning on TIMELY for RDMA transfers on a fabric with PFC (Priority Flow Control) lowers 99 percentile tail latency by $9X$. This tail latency is $13X$ lower than that of DCTCP running in an optimized kernel.

2. THE VALUE OF RTT AS A CONGESTION SIGNAL IN DATACENTERS

Existing datacenter transports use signals from network switches to detect the onset of congestion and run with low levels of latency and loss [15,35,37]. We argue that network queueing delay derived from RTT measurements, without the need for any switch support, is a superior congestion signal.

RTT directly reflects latency. RTTs are valuable because they directly measure the quantity we care about: end-to-end latency inflated by network queueing. Signals derived from queue occupancy such as ECN fail to directly inform this metric. An ECN mark on a packet simply indicates that the queue measure corresponding to the packet exceeds a threshold. Rich use of QoS in the datacenter means it is not possible to convert this threshold into a single corresponding latency. Multiple queues with different priorities share the same output link, but the ECN mark only provides information about those with occupancy exceeding a threshold. Low priority traffic can experience large queueing delays without necessarily building up a large queue. In such circumstances, queueing delay reflects the state of congestion in the network which is not reflected by queue occupancy of low priority traffic. Further, an ECN mark describes behavior at a single switch. In a highly utilized network, congestion occurs at multiple switches, and ECN signals cannot differentiate among them. The RTT accumulates information about the end-to-end path. It includes the NIC, which may also become congested but is ignored by most schemes. Finally, RTTs work even for the lossless network fabrics commonly used to support FCoE [2]. In these fabrics, mere queue occupancy can fail to reflect congestion because of Priority Flow Control mechanisms used to ensure zero packet loss.

RTT can be measured accurately in practice. A key practical hurdle is whether RTTs can be measured accurately in datacenters where they are easily 1000X smaller than wide-area latencies. Many factors have precluded accurate measurement in the past: variability due to kernel scheduling; NIC performance techniques including offload (GSO/TSO, GRO/LRO); and protocol processing such as TCP delayed acknowledgements. This problem is severe in datacenters where each of these factors is large enough to overwhelm propagation and queuing delays.

Fortunately, recent NICs provide hardware support to solve these problems [1,3,5,8,9] and can accurately record the time of packet transmission and reception without being affected by software-incurred delays. These methods must be used with care lest they overly tax the NIC. We describe our use of NIC support later in the paper. These NICs also provide hardware-based acknowledgements for some protocols.

The combination of these features lets us take precise RTT measurements to accurately track end-to-end network

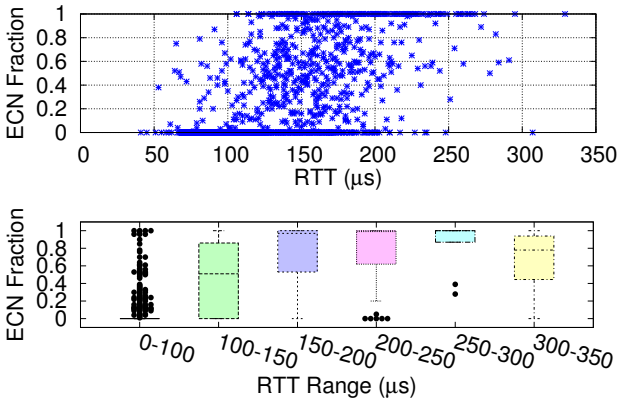

Figure 3: Fraction of packets with ECN marks versus RTTs shown as scatter plot (top) and box plot (bottom).

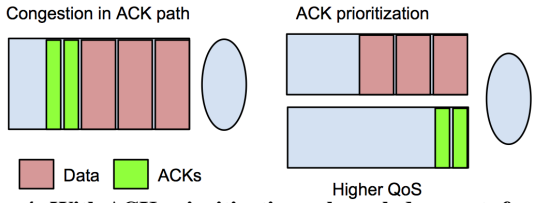

Figure 4: With ACK prioritization acknowledgements from the primary incast are not delayed by the data of the secondary incast.

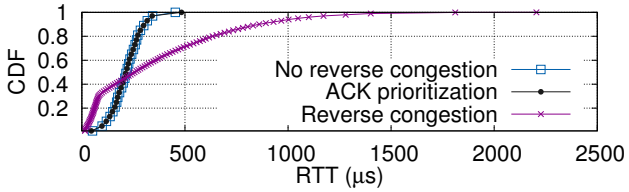

Figure 5: In the presence of reverse congestion, RTT measurements with ACK prioritization are indistinguishable from RTTs that do not experience any reverse path congestion.

queues. The following experiment shows this behavior: we connect two hosts to the same network via 10 Gbps links and send 16 KB ping-pong messages without any cross traffic and on a quiescent network. Since there is no congestion, we expect the RTT measurements to be low and stable. Figure 1 compares the CDF of RTTs measured using NIC hardware timestamps and RTTs measured via the OS TCP stack. The CDF for RTTs using NIC timestamps is nearly a straight line, indicating small variance. In contrast, the RTTs measured by kernel TCP are larger and much more variable.

RTT is a rapid, multi-bit signal. Network queueing delay can be calculated by subtracting known propagation and serialization delays from an RTT. Surprisingly, we have found that this method provides richer and faster information about the state of network switches than explicit network switch signals such as ECN marks. As a binary quantity, ECN marks convey a single bit of information, while RTT measurements convey multiple bits and capture the full extent of end to end queueing delay aggregated across multiple switches rather than the presence of queueing exceeding a simple fixed threshold at any one of N switches. Since each packet may carry an ECN mark, it is plausible that the sequence of marks can close this gap and convey multi-bit information about the congestion level, as is done in DCTCP (which in practice is how ECN is used in datacenters). However, modern practices such as 64 KB segmentation offload undermine the independence of marks made closely in time because host sources are bursty. Large bursts at wire speed tend to see most packets either above or below the marking threshold.

Assuming it is accurate, a single high RTT measurement immediately signals the extent of congestion. This RTT measurement works even with packet bursts for a flow sent along a single network path: the RTT of the last packet in the burst tracks the maximum RTT across packets since delays to earlier packets push out later packets. To show how well RTT tracks network queuing delay, we set up an incast experiment with 100 flows on 10 client machines simultaneously transmitting to a single server. To incorporate NIC offload, we send 64 KB messages and collect only a single RTT measurement per message on the client side. The bottleneck link is a 10 Gbps link to the server. We sample the switch queue each microsecond.

Figure 2 shows the CDF of RTTs as measured at the end system compared to the queue occupancy measured directly at the switch and shown in units of time computed for a 10 Gbps link. The two CDFs match extremely well.

In contrast, the ECN signal does not correlate well with RTT and hence with the amount of queuing. We set up a similar incast experiment, except now TCP senders perform long transfers to a single receiver. The bottleneck is the 10 Gbps link to the receiver, with an 80 KB switch ECN marking threshold. We instrument the senders to measure the fraction of packets received with ECN marks in every round-trip time, using the advance of *SND_UNA* [41] to mark the end of an RTT round. We also measure the minimum RTT sample in every round, as prior work has found it to be a robust descriptor untainted by delayed ACKs and send/receive offload schemes. Both the scatter and box plots [1] [4] in Figure 3 show only a weak correlation between the fraction of ECN marks and RTTs.

Limitations of RTTs. While we have found the RTT signal valuable, an effective design must use it carefully. RTT measurements lump queueing in both directions along the network path. This may confuse reverse path congestion experienced by ACKs with forward path congestion experienced by data packets. One simple fix is to send ACKs with higher priority, so that they do not incur significant queuing delay. This method works in the common case of flows that predominantly send data in one direction; we did not need more complicated methods.

We conducted an experiment to verify the efficacy of ACK prioritization: we started two incasts (*primary* and *secondary*) such that the ACKs of the primary incast share the same congested queue as the data segments of the secondary incast, as shown in Figure 4. Figure 5 shows the CDF of

[1]Whiskers extend to the most distant point within $1.2X$ interquartile range. Points outside these limits are drawn individually.

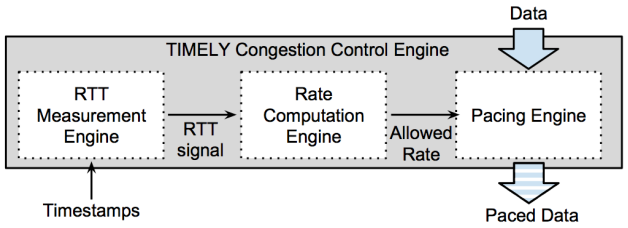

Figure 6: TIMELY overview.

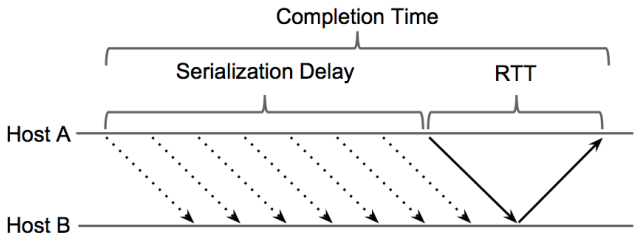

Figure 7: Finding RTT from completion time.

RTTs from the primary incast for the following three cases: 1) no congestion in ACK path (no secondary incast); 2) in the presence of congestion in ACK path; and 3) ACKs from primary incast are prioritized to higher QoS queue in the presence of reverse congestion. We find that the reverse congestion creates noise in RTT measurements of the primary incast and elongates the tail latencies. Throughput of the primary incast is also impacted (and hence the smaller RTTs in the lower percentiles). With ACK prioritization, the RTTs measured in the primary incast are indistinguishable from those measured in the absence of reverse path congestion.

In future work, it would be straightforward to calculate variations in one-way delay between two packets by embedding a timestamp in a packet (e.g., TCP timestamps). The change in queuing delay is then the change in the arrival time minus send time of each packet. This method needs only clocks that run at the same rate, which is a much less stringent requirement than synchronized clocks.

The other classic shortcoming of RTTs is that changing network paths have disparate delays. It is less of a problem in datacenters as all paths have small propagation delays.

3. TIMELY FRAMEWORK

TIMELY provides a framework for rate control that is independent of the transport protocol used for reliability. Figure 6 shows its three components: 1) RTT measurement to monitor the network for congestion; 2) a computation engine that converts RTT signals into target sending rates; and 3) a control engine that inserts delays between segments to achieve the target rate. We implement TIMELY in host software with NIC hardware support, and run an independent instance for each flow.

3.1 RTT Measurement Engine

We assume a traditional transport where the receiver explicitly ACKs new data so that we may extract an RTT. We define the RTT in terms of Figure 7, which shows the timeline of a message: a segment consisting of multiple packets is sent as a single burst and then ACKed as a unit by the receiver. A *completion event* is generated upon receiving an ACK for a segment of data and includes the ACK receive time. The time from when the first packet is sent ($\mathrm{t_{send}}$) until the ACK is received ($\mathrm{t_{completion}}$) is defined as the *completion time*. Unlike TCP, there is one RTT for the set of packets rather than one RTT per 1-2 packets. There are several delay components: 1) the serialization delay to transmit all packets in the segment, typically up to 64 KB; 2) the round-trip wire delay for the segment and its ACK to propagate across the datacenter; 3) the turnaround time at the receiver to generate the ACK; and 4) the queuing delay at switches experienced in both directions.

We define the RTT to be the propagation and queueing delay components only. The first component is a deterministic function of the segment size and the line rate of the NIC. We compute and subtract it from the total elapsed time so that the RTTs input to TIMELY's rate computation engine are independent of segment size. The third component is sufficiently close to zero in our setting with NIC-based ACKs that we can ignore it. Of the remaining components, the second is the propagation delay including the packet-based store-and-forward behavior at switches. It is the minimum RTT and fixed for a given flow. Only the last component – the queueing delay – causes variation in the RTT, and it is our focus for detecting congestion. In summary, TIMELY running on Host A (shown in Figure 7) computes RTT as:

$$RTT = \mathrm{t_{completion}} - \mathrm{t_{send}} - \frac{\text{seg. size}}{\text{NIC line rate}}$$

For context, in a 10 Gbps network, serialization of 64 KB takes 51 μs, propagation may range from 10-100 μs, and 1500 B of queuing takes 1.2 μs. We rely on two forms of NIC support to precisely measure segment RTTs, described next.

ACK Timestamps. We require the NIC to supply the completion timestamp, $\mathrm{t_{completion}}$. As shown in §2, OS timestamps suffer from variations such as scheduling and interrupts that can easily obscure the congestion signal. $\mathrm{t_{send}}$ is the NIC hardware time that's read by host software just before posting the segment to NIC.

Prompt ACK generation. We require NIC-based ACK generation so that we can ignore the turnaround time at the receiver. An alternative would be to use timestamps to measure the ACK turnaround delay due to host processing delay. We have avoided this option because it would require augmenting transport wire formats to include this difference explicitly.

Fortunately, some modern NICs [1, 3, 5, 8, 9] provide one or both features, and our requirements are met naturally with a messaging implementation that timestamps segment acknowledgement. A specific implementation of TIMELY in the context of RDMA is described in §5. We believe our design is more generally applicable to TCP with some care to work with the batching behavior of the NIC, correctly associate an ACK with the reception of new data, and compensate for ACK turnaround time.

3.2 Rate Computation Engine

This component implements our RTT-based congestion control algorithm as detailed in §4. The interface to the rate computation engine is simple. Upon each completion event, the RTT measurement engine provides the RTT in microseconds to the rate computation engine. While this is the only required input, additional timing information could also be useful, e.g., the delay incurred in the NIC. There is no requirement for packet-level operation; in normal operation we expect a single completion event for a segment of size up to 64 KB due to NIC offload. The rate computation engine runs the congestion control algorithm upon each completion event, and outputs an updated target rate for the flow.

3.3 Rate Control Engine

When a message is ready to be sent, the rate control engine breaks it into segments for transmission, and sends each segment to the scheduler in turn. For runtime efficiency, we implement a single scheduler that handles all flows. The scheduler uses the segment size, flow rate (provided by the rate computation engine), and time of last transmission to compute the send time for the current segment with the appropriate pacing delay. The segment is then placed in a priority queue in the scheduler. Segments with send times in the past are serviced in round-robin fashion; segments with future send times are queued. After the pacing delay has elapsed, the rate control engine passes the segment to the NIC for immediate transmission as a burst of packets. Data is first batched into 64 KB segments, following which the scheduler computes the pacing delay to insert between two such batched segments. Note that 64 KB is the maximum batching size and is not a requirement, e.g., the segment sizes for a flow that only has small messages to exchange at any given time will be smaller than 64 KB. We later present results for segment sizes smaller than 64 KB as well.

TIMELY is rate-based rather than window-based because it gives better control over traffic bursts given the widespread use of NIC offload. The bandwidth-delay product is only a small number of packet bursts in datacenters, e.g., 51 μs at 10 Gbps is one 64 KB message. In this regime, windows do not provide fine-grained control over packet transmissions. It is easier to directly control the gap between bursts by specifying a target rate. As a safeguard, we limit the volume of outstanding data to a static worst-case limit.

4. TIMELY CONGESTION CONTROL

Our congestion control algorithm runs in the rate computation engine. In this section, we describe our environment and key performance metrics, followed by our gradient-based approach and algorithm.

4.1 Metrics and Setting

The datacenter network environment is characterized by many bursty message workloads from tightly-coupled forms of computing over high bandwidth, low-latency paths. It is the opposite of the traditional wide-area Internet in many respects. Bandwidth is plentiful, and it is flow completion time (e.g., for a Remote Procedure Call (RPC)) that is the overriding concern. For short RPCs, the minimum completion time is determined by the propagation and serialization delay. Hence, we attempt to minimize any queueing delay to keep RTTs low. The latency tail matters because application performance degrades when even a small fraction of the packets are late [21]. Consistent low-latency implies low queueing delay and near zero packet loss, since recovery actions may greatly increase message latency. Longer RPCs will have larger completion times because of the time it takes to transmit more data across a shared network. To keep this added time small, we must maintain high aggregate throughput to benefit all flows and maintain approximate fairness so that no one flow is penalized.

Our primary metrics for evaluation are tail (99th percentile) RTT and aggregate throughput, as they determine how quickly we complete short and long RPCs (assuming some fairness). When there is a conflict between throughput and packet RTT, we prefer to keep RTT low at the cost of sacrificing a small amount of bandwidth. This is because bandwidth is plentiful to start with, and increased RTT directly impacts the completion times of short transfers. In effect, we seek to ride the throughput/latency curve to the point where tail latency becomes unacceptable. Secondary metrics are fairness and loss. We report both as a check rather than study them in detail. Finally, we prefer a stable design over higher average, but oscillating rates for the sake of predictable performance.

4.2 Delay Gradient Approach

Delay-based congestion control algorithms such as FAST TCP [46] and Compound TCP [44] are inspired by the seminal work of TCP Vegas [16]. These interpret RTT increase above a baseline as indicative of congestion: they reduce the sending rate if delay is further increased to try and maintain buffer occupancy at the bottleneck queue around some predefined threshold. However, Kelly et al. [33] argue that it is not possible to control the queue size when it is shorter in time than the control loop delay. This is the case in datacenters where the control loop delay of a 64 KB message over a 10 Gbps link is at least 51 μs, and possibly significantly higher due to competing traffic, while one packet of queuing delay lasts 1 μs. The most any algorithm can do in these circumstances is to control the *distribution* of the queue occupancy. Even if controlling the queue size were possible, choosing a threshold for a datacenter network in which multiple queues can be a bottleneck is a notoriously hard tuning problem.

TIMELY's congestion controller achieves low latencies by reacting to the *delay gradient* or derivative of the queuing with respect to time, instead of trying to maintain a standing queue. This is possible because we can accurately measure differences in RTTs that indicate changes in queuing delay. A positive delay gradient due to increasing RTTs indicates a rising queue, while a negative gradient indicates a receding queue. By using the gradient, we can react to queue growth without waiting for a standing queue to form – a strategy that helps us achieve low latencies.

Algorithm 1: TIMELY congestion control.

Data: new_rtt
Result: Enforced rate
new_rtt_diff = new_rtt - prev_rtt ;
prev_rtt = new_rtt ;
rtt_diff = (1 - α) · rtt_diff + α · new_rtt_diff ;
▷ α: EWMA weight parameter
normalized_gradient = rtt_diff / minRTT ;
if *new_rtt* < T_{low} **then**
  rate ← rate + δ ;
  ▷ δ: additive increment step
  return;
if *new_rtt* > T_{high} **then**
  rate ← rate · $\left(1 - \beta \cdot \left(1 - \frac{T_{high}}{new\_rtt}\right)\right)$;
  ▷ β: multiplicative decrement factor
  return;
if *normalized_gradient* ≤ 0 **then**
  rate ← rate + N · δ ;
  ▷ N = 5 if gradient<0 for five completion events (HAI mode); otherwise N = 1
else
  rate ← rate · (1 - β · normalized_gradient)

Delay gradient is a proxy for the rate mismatch at the bottleneck queue. We are inspired by RCP, XCP, PI, and QCN [10, 28, 32, 39] that find explicit feedback on the rate mismatch has better stability and convergence properties than explicit feedback based only on queue sizes; the latter can even cause the queue to be *less* accurately controlled. The key difference is that all of these prior controllers operate at point queues in the network, while TIMELY achieves similar benefits by using the end-to-end delay gradient.

The model we assume is N end hosts all sending data at a total rate $y(t)$ into a bottleneck queue with drain rate C, i.e. the outgoing rate is $\leq C$. We denote the queuing delay through the bottleneck queue by $q(t)$. If $y(t) > C$, the rate at which the queue builds up is $(y(t) - C)$. Since queued data drains at a rate C, the queuing delay gradient is given by $\frac{dq(t)}{dt} = \frac{(y(t)-C)}{C}$. The gradient is dimensionless. It is positive for $y(t) > C$ and signals how quickly the queue is building. The negative gradient when $y(t) < C$, signals how quickly the queue is draining. Hence, the delay gradient measured through RTT signals acts as an indicator for the rate mismatch at the bottleneck. This reasoning holds as long as there is some non-zero queue in the network. When there is zero queueing or queues are not changing in size, the measured gradient is also zero. TIMELY strives to match the aggregate incoming rate $y(t)$ to the drain rate, C, and so adapts its per-connection rate, $R(t)$, in proportion to the measured error of $\frac{(y(t)-C)}{C} = \frac{dq(t)}{dt} = \frac{d(RTT)}{dt}$.

4.3 The Main Algorithm

Algorithm 1 shows pseudo-code for our congestion control algorithm. TIMELY maintains a single rate $R(t)$ for each connection and updates it on every completion event

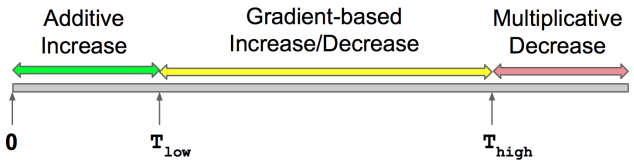

Figure 8: Gradient tracking zone with low and high RTT thresholds.

using RTT samples. It employs gradient tracking, adjusting the rate using a smoothed delay gradient as the error signal to keep throughput close to the available bandwidth. Additionally, we employ thresholds to detect and respond to extreme cases of under utilization or overly high packet latencies. Figure 8 shows the gradient zone along with the two thresholds. When the RTT is in the nominal operating range, the gradient tracking algorithm computes the delay gradient from RTT samples and uses it to adjust the sending rate.

Computing the delay gradient. We rely on accurate RTT measurements using NIC timestamps (§3). To compute the delay gradient, TIMELY computes the difference between two consecutive RTT samples. We normalize this difference by dividing it by the minimum RTT, obtaining a dimensionless quantity. In practice, the exact value of the minimum RTT does not matter since we only need to determine if the queue is growing or receding. We therefore use a fixed value representing the wire propagation delay across the datacenter network, which is known ahead of time. Finally, we pass the result through an EWMA filter. This filter allows us to detect the overall trend in the rise and fall in the queue, while ignoring minor queue fluctuations that are not indicative of congestion.

Computing the sending rate. Next, TIMELY uses the normalized gradient to update the target rate for the connection. If the gradient is negative or equals zero, the network can keep up with the aggregate incoming rate, and therefore there is room for a higher rate. In this case, TIMELY probes for more bandwidth by performing an additive increment for the connection: $R = R + \delta$, where δ is the bandwidth additive increment constant. When the gradient is positive, the total sending rate is greater than network capacity. Hence, TIMELY performs a multiplicative rate decrement β, scaled by the gradient factor:

$$R = R\left(1 - \beta \frac{d(RTT(t))}{dt}\right)$$

The delay gradient signal, which is based on the total incoming and outgoing rates, is common for all connections along the same congested path. The well-known AIMD property ensures that our algorithm achieves fairness across connections [19]. Connections sending at a higher rate observe a stronger decrease in their rate, while the increase in rate remains same for all connections.

While the delay gradient is effective in normal operation, situations with significant under-utilization or high latency require a more aggressive response. Next we discuss how TIMELY detects and responds to these situations.

Need for RTT low threshold T_{low}. The ideal environment for our algorithm is one where packets are perfectly paced.

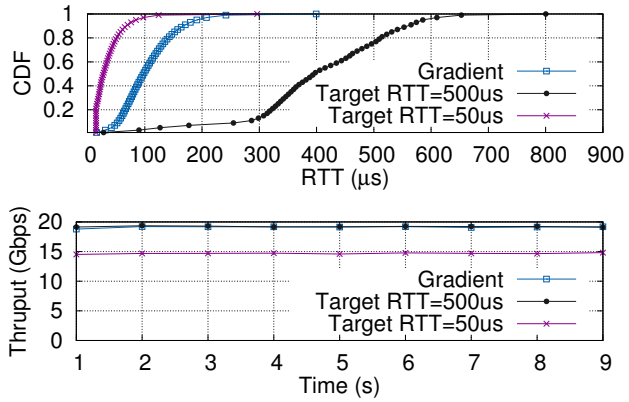

Figure 9: Comparison of gradient (low and high thresholds of 50 μs and 500 μs) with target-based approach (T_{target} of 50 μs and 500 μs).

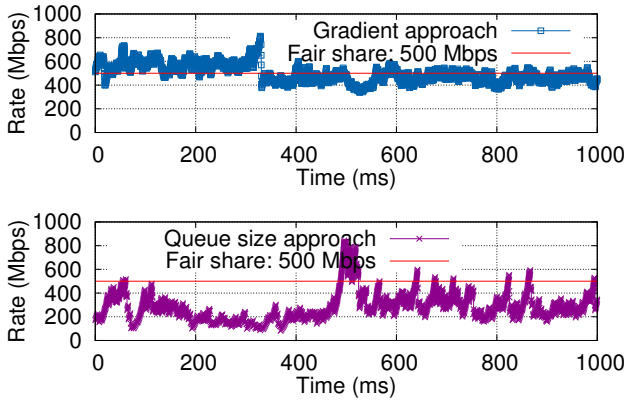

Figure 10: Per-connection rates in the gradient approach are smooth (top) while those in the queue-size based approach (with T_{target} = 50μs) are more oscillatory (bottom).

However, in practical settings, the TIMELY rate is enforced on a segment granularity that can be as large as 64 KB. These large segments lead to packet bursts, which result in transient queues in the network and hence RTT spikes when there is an occasional collision. Without care, the core algorithm would detect a positive gradient due to a sudden RTT spike and unnecessarily infer congestion and back-off. We avoid this behavior by using a low threshold T_{low} to filter RTT spikes; the adjustment based on delay gradient kicks in for RTT samples greater than the threshold. T_{low} is a (nonlinear) increasing function of the segment size used in the network, since larger messages cause more bursty queue occupancy. We explore this effect in our evaluation, as well as show how fine-grained pacing at the hosts can reduce burstiness and hence the need for a low threshold.

Need for RTT high threshold T_{high}. The core gradient algorithm maintains close to the bottleneck link throughput while building very little queue. However, in theory, it is possible for the gradient to stay at zero while the queue remains at a high, fixed level. To remove this concern, T_{high} serves as an upper bound on the tolerable end-to-end network queue delay, i.e., the tail latency. It provides a way to reduce the rate independent of gradient value if the latency grows, a protection that is possible because we operate in a datacenter environment with known characteristics. If the measured RTT is greater than T_{high}, we reduce the rate multiplicatively:

$$R = R\Big(1 - \beta\Big(1 - \frac{T_{high}}{RTT}\Big)\Big)$$

Note that we use the instantaneous rather than smoothed RTT. While this may seem unusual, we can slow down in response to a single overly large RTT because we can be confident that it signals congestion, and our priority is to maintain low packet latencies and avoid loss. We tried responding to average RTT as a congestion indicator, and found that it *hurts* packet latency because of the extra delay in the feedback loop. By the time the average rose, and congestion control reduces the rate, queueing delay has already increased in the network. Our finding is inline with [27] which shows through a control theoretic analysis that the averaged queue length is a failing of RED AQM. We show in §6 how T_{high} lets us ride to the right along the throughput-delay tradeoff curve.

Hyperactive increase (HAI) for faster convergence. Inspired by the *max probing* phase in TCP BIC, CUBIC [42] congestion control, and QCN [10], we include an *HAI* option for faster convergence as follows: if TIMELY does not reach the new fair share after a period of slow growth, i.e., the gradient is negative for several consecutive completion times, then HAI switches to a faster additive increase in which the rate is incremented by $N\delta$ instead of δ. This is useful when the new fair share rate has dramatically increased due to reduced load in the network.

4.4 Gradient versus Queue Size

We highlight through an experiment how the gradient approach differs from a queue size based scheme. If we set the same value for T_{low} and T_{high}, then TIMELY congestion control reduces to a queue size based approach (similar to TCP FAST algorithm; FAST in turn is a scaled, improved version of Vegas). Denote T_{target} as the value of this single RTT threshold, i.e., $T_{target} = T_{low} = T_{high}$. Then the rate is increased additively and decreased multiplicatively with the decrease factor scaled by the queue size above T_{target}.

Figure 9 compares the RTT and throughput for the gradient and queue size based approach for an incast traffic pattern. (See §6 for experimental details.) We use low and high thresholds of 50 μs and 500 μs for gradient, versus T_{target} of 50 μs and 500 μs for the queue-sized approach. We see that the queue size approach can maintain either low latency or high throughput, but finds it hard to do both. By building up a standing queue up to a high T_{target} of 500 μs, throughput is optimized, but at the cost of latency due to queueing. Alternatively, by keeping the standing queue at a low T_{target} of 50 μs, latency is optimized, but throughput suffers as the queue is sometimes empty. By operating on the rising and falling queue, the gradient approach predicts the onset of congestion. This lets it deliver the high throughput of a high queue target while keeping the tail latency close to that of a low target.

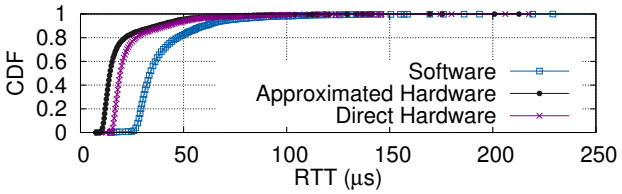

Figure 11: Comparison of the accuracy of NIC and SW timestamps.

Furthermore, as shown in Figure 10, the connection rates oscillate more in the queue-size approach, as it drives the RTT up and down towards the target queue size. The gradient approach maintains a smoother rate around the fair share. Similar results are shown in control theory terms for AQMs using the queue-size approach and gradient approach [28].

The main take-away is that T_{low} and T_{high} thresholds effectively bring the delay within a target range and play a role similar to the target queue occupancy in many AQM schemes. Using the delay gradient improves stability and helps keep the latency within the target range.

5. IMPLEMENTATION

Our implementation is built on 10 Gbps NICs with OS-bypass capabilities. The NICs support multi-packet segments with hardware-based ACKs and timestamps. We implemented TIMELY in the context of RDMA (Remote Direct Memory Access) as a combination of NIC functionality and host software. We use RDMA primitives to invoke NIC services and offload complete memory-to-memory transfers to the NIC. In particular, we mainly use RDMA Write and Read to take a message from local host memory and send it on the network as a burst of packets. On the remote host, the NIC acknowledges receipt of the complete message and places it directly in the remote memory for consumption by the remote application. The local host is notified of the acknowledgement when the transfer is complete. We describe below some of the notable points of the implementation.

Transport Interface. TIMELY is concerned with the congestion control portion of the transport protocol; it is not concerned with reliability or the higher-level interface the transport exposes to applications. This allows the interface to the rest of the transport to be simple: message send and receive. When presented with a message at the sender, TIMELY breaks it into smaller segments if it is large and sends the segments at the target rate. A message is simply an ordered sequence of bytes. The segment is passed to the NIC and then sent over the network as a burst of packets. On the remote host, the NIC acknowledges receipt of the complete segment. At the receiver, when a segment is received it is passed to the rest of the transport for processing. This simple model supports transports ranging from RPCs to bytestreams such as TCP.

Using NIC Completions for RTT Measurement. In practice, using NIC timestamps is challenging. Our NIC only records the absolute timestamp of when the multi-packet operation finishes and therefore our userspace software needs to record a timestamp of when the operation was posted to the NIC. This requires a scheme to map host clock to NIC clock, as well as calibration. We record host (CPU) timestamps when posting work to the NIC and build a calibration mechanism to map NIC timestamps to host timestamps. A simple linear mapping is sufficient. The mechanism works well because the probability of being interrupted between recording the host send timestamp and actually handing the message to the NIC is fairly low. Figure 11 compares RTTs obtained from NIC HW timestamps, the calibration mechanism, and pure software only timestamps. Note that TIMELY does not spin, so interrupts and wakeups are included in the software timestamp numbers. It clearly demonstrates that the calibration mechanism is just as accurate as using only NIC timestamps. Furthermore, the SW timestamps have a large variance, which increases as load on the host increases.

We consider any NIC queuing occurring to be part of the RTT signal. This is important because NIC queuing is also indicative of congestion and is handled by the same rate-based controls as network queueing — even if the NIC were to give us an actual send timestamp, we would want the ability to observe NIC queuing.

RDMA rate control. For RDMA Writes, TIMELY on the sender directly controls the segment pacing rate. For RDMA Reads, the receiver issues read requests, in response to which the remote host performs a DMA of the data segments. In this case, TIMELY cannot directly pace the data segments, but instead achieves the same result by pacing the read requests: when computing the pacing delay between read requests, the rate computation engine takes into account the data segment bytes read from the remote host.

Application limited behavior. Applications do not always have enough data to transmit for their flows to reach the target rate. When this happens, we do not want to inadvertently increase the target rate without bound because the network appears to be uncongested. To prevent this problem, we let the target rate increase only if the application is sending at more than 80% of the target rate, and we also cap the maximum target rate at 10 Gbps. The purpose of allowing some headroom is to let the application increase its rate without an unreasonable delay when it does have enough data to send.

Rate update frequency. TIMELY's rate update equation assumes that there is at most one completion event per RTT interval. The transmission delay of a 64 KB message on a 10 Gbps link is 51 μs; with a minimum RTT of 20 μs, there can be at most one completion event in any given minimum RTT interval. However, for small segment sizes, there can be multiple completion events within a minimum RTT interval. In such a scenario, we want to update the rate based on the most recent information. We do so by updating the rate for every completion event, taking care to scale the updates by the number of completions per minimum RTT interval so that we do not overweigh the new information.

For scheduler efficiency, the rate control engine enforces rate updates lazily. When the previously computed send time for a segment elapses, the scheduler checks the current rate. If the rate has decreased, we recompute the send time, and

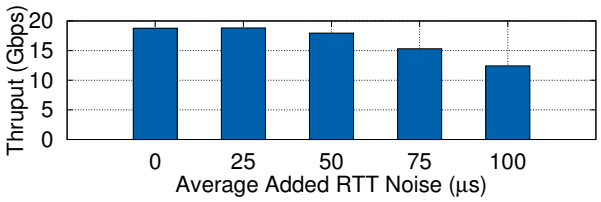

Figure 12: Impact of RTT noise on TIMELY throughput.

re-queue the segment if appropriate. Otherwise, the scheduler proceeds to send the segment to the NIC.

Additional pacing opportunities. By default, the NIC sends a segment as a burst of packets at the link line rate. We explore another possibility: using NIC rate-limiters to transmit a burst of packets at less than the line rate. The rationale is to supplement the pacing engine that spreads packets of the flow over time with NIC-sized units of work. With hardware rate-limiters [43], it is feasible to offload part of the responsibility for pacing to the NIC. However due to hardware constraints, re-configuring pacing rate every few RTTs is not always feasible. Instead, we use a hybrid approach: software pacing of large segments and hardware pacing at fixed rate below the link rate, e.g. 5 Gbps on a 10 Gbps link. At these high-rates, the purpose of NIC pacing is to insert gaps in the burst so that multiple bursts mix at switches without causing latency spikes. In this case, the rate control engine compensates for the NIC pacing delays by treating it as a lower transmission line rate.

6. EVALUATION

We evaluate a real host-based implementation of TIMELY at two scales. First, we examine the basic properties of the congestion controller such as throughput, fairness, packet latency, and timing accuracy in an incast setting. For these microbenchmarks, we use a small-scale testbed with a rack of equipment. Second, we run TIMELY on a larger scale testbed of a few hundred machines in a classic Clos network topology [12, 34]. Along with running the traffic workload, hosts collect measurements of per-connection throughputs, RPC latencies, and RTTs (we established in §2 that host RTTs correspond well with queueing delays measured at the switches). All links are 10 Gbps unless mentioned otherwise. The OS used in all experiments is Linux.

To place our results in context, we compare TIMELY with two alternatives. First, we use OS-bypass messaging over a fabric with Priority Flow Control (PFC) as commonly used for low loss and latency in FCoE, e.g. DCB [2]. The RDMA transport is in the NIC and sensitive to packet drops, so PFC is necessary because drops hurt performance badly. We add TIMELY to this RDMA setting to observe its benefits; we check that pause message counts are low to verify that there is sufficient switch buffering for TIMELY to work and PFC is not an inadvertent factor in our experimental results. Second, we compare against an optimized kernel stack that implements DCTCP [35] running on the same fabric without the use of PFC. We choose DCTCP as a point of comparison because it is a well-known, modern datacenter transport that has been deployed and proven at scale.

Metric	DCTCP	FAST*			PFC	TIMELY
		10M	50M	100M		
Total Throughput (Gbps)	19.5	7.5	12.5	17.5	19.5	19.4
Avg. RTT (us)	598	19	120	354	658	61
99-percentile RTT (us)	1490	49	280	460	1036	116

Table 1: Overall performance comparison. FAST* is shown with network buffered traffic parameter in Mbps.

Henceforth we refer to: 1) *DCTCP*, for kernel DCTCP over a fabric without PFC; 2) *PFC*, for OS-bypass messaging over a fabric with PFC; 3) *FAST**, for OS-bypass messaging with TCP FAST-like congestion control algorithm; and 4) *TIMELY*, for OS-bypass messaging with TIMELY over a fabric with PFC

Unless mentioned otherwise, we use the following parameters for TIMELY: segment size of 16 KB, T_{low} of 50 μs, T_{high} of 500 μs, additive increment of 10 Mbps, and a multiplicative decrement factor (β) of 0.8.

6.1 Small-Scale Experiments

We use an incast traffic pattern for small-scale experiments (unless otherwise specified) since it is a key congestion scenario for datacenter networks [45]. To create incast, 10 client machines on a single rack send to a single server on the same rack. Each client runs 4 connections, i.e., 40 total concurrent connections. Each connection sends 16 KB segments at a high enough aggregate rate to saturate the server bandwidth of 2x10G link which is the bottleneck for the experiment. This is a demanding workload for testing congestion control: while there are many connections present in the datacenter, the number of connections limited by network capacity is normally small.

Required RTT measurement accuracy. To evaluate the accuracy of RTT samples required by TIMELY, we add noise to the measured RTTs and observe the impact on throughput. We add random noise uniformly distributed in the range of $[0, x]$ μs to each RTT sample, where x is set to 0, 50, 100, 150, 200. Figure 12 shows the total throughput measured on the server at different noise levels. Average noise of 50 μs causes visible throughput degradation, and higher noise leads to more severe performance penalties. A T_{low} value lower than 50 μs lowers the tolerance to RTT noise even further. Note that this level of noise is easily reachable by software timestamping (due to scheduling delays, coalescing, aggregation, etc.). Hence, accurate RTT measurement provided by NIC support is the cornerstone of TIMELY.

Comparison with PFC. Table 1 compares TIMELY with OS-bypass messaging over a fabric with conventional RDMA deployment over PFC. While the throughput is slightly lower with TIMELY, the median and tail RTTs are lower by more than order of magnitude and pauses are not triggered at all. Next, we break out connections to show that TIMELY also delivers excellent performance for individual connections. Figure 13 shows a timeline of the observed RTT and throughput for a sample for four individual connections using TIMELY. Each datapoint represents a single completion event. The fair share for each connection is 500 Mbps. We see that the throughput is close to the

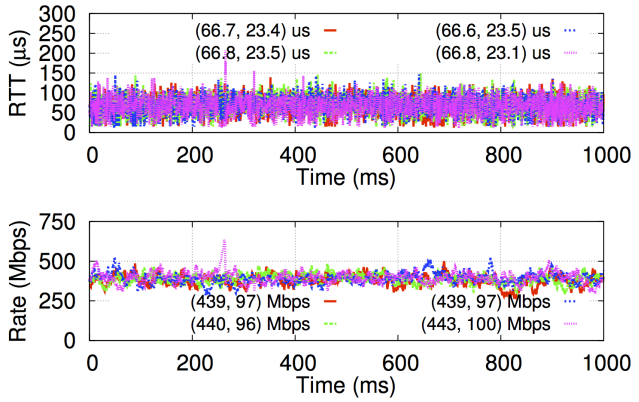

Figure 13: RTTs and sending rates of a sample of connections for TIMELY. The legend gives the mean and standard deviation for each connection.

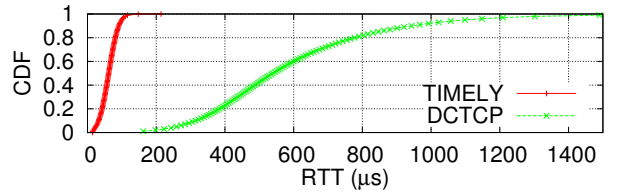

Figure 14: CDF of RTT distribution

fair share and the RTT remains consistently low. To quantify the fairness of bandwidth allocation across connections, we compute a Jain fairness index [31] of 0.953 for TIMELY and 0.909 for PFC. Our design is more fair, and has a high enough index to meet our fairness needs.

Comparison with DCTCP. We compare TIMELY with DCTCP. To review briefly: senders emit ECN capable packets; switches mark each packet queued beyond a fixed threshold; receivers return all ECN marks to the sender; and senders adapt their rate based on the fraction of packets with ECN marks in a window. Note that of necessity our comparison is for two different host software stacks, as DCTCP runs in an optimized kernel without PFC support whereas TIMELY is used with OS-bypass messaging. We did not implement DCTCP in OS-bypass environment due to NIC firmware limitations on processing ECN feedback [48]. We set the switch ECN marking threshold to $K = 80$ KB. This is less than the DCTCP author recommendation of $K = 65$ packets for 10 Gbps operation as we are willing to sacrifice a small amount of throughput to ensure consistently low latency. Table 1 summarizes the results averaged across three runs of ten minutes, with the RTT distribution shown in Figure 14. TIMELY keeps the average end-to-end RTT 10X lower than DCTCP (60 μs vs. 600 μs). More significantly, the tail latency drops by almost 13X (116 μs vs. 1490 μs). No loss nor PFC packets were observed. These latency reductions do not come at the cost of throughput.

Comparison with TCP FAST-like algorithm. We next implement FAST* using TCP FAST congestion control design [46] in place of Algorithm 1. Instead of periodically updating the congestion window, we use the TCP FAST equation to adjust the pacing rate. Note that TCP FAST is tunable

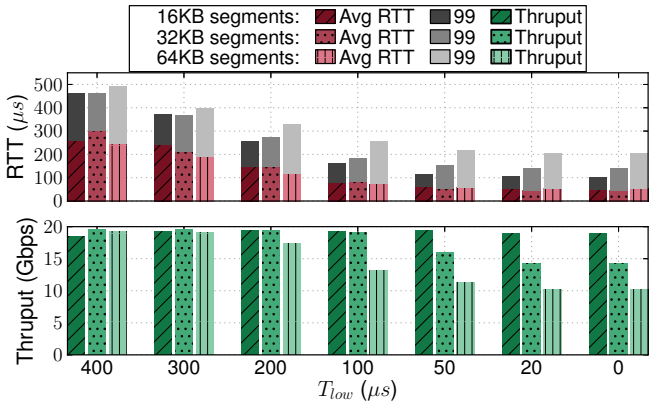

Figure 15: Throughput and RTT varying with T_{low} and segment size

by a protocol parameter α that controls the balance between fairness and the amount of buffered traffic in the network. In [46], α is the number of packets of each flow pending in the network. We convert α to a throughput value since TIMELY is rate-based. Table 1 shows our results with three different values of α. For a small α = 10 Mbps, FAST* can achieve low tail latency 49us albeit at a significant throughput loss – achieving only 7.5 Gbps out of 20 Gbps line rate. At larger values (α = 50 Mbps and 100 Mbps), TIMELY still achieves better throughput and latency trade-offs.

Varying T_{low}. Our performance is influenced by algorithm parameters, which we explore starting with the low threshold. Our purpose is to highlight factors that affect the thresholds, not to tune parameters. By design no more than a default setting is necessary. The low threshold exists to absorb the RTT variation during uncongested network use. The expected variation is related to the maximum segment size, since as segments grow larger the effect on the RTT of occasional segment collisions increases. Figure 15 shows how the bottleneck throughput and RTT times vary with different values of T_{low} for segments of size 16 KB, 32 KB and 64 KB.

We see that decreasing the low threshold reduces the network delay. This is because a lower threshold allows the use of the RTT gradient more often to modulate the rate in response to queue build-ups. But lower thresholds eventually have an adverse effect on throughput. This is best seen with bursty traffic. For 16 KB segment size, when the burstiness is relatively low, a T_{low} of just 50 μs gives us the highest throughput (19.4 Gbps), with the throughput being only slightly lower (18.9 Gbps) without any T_{low}. However, as we increase the segment size and hence the burstiness, the throughput falls quickly when the threshold becomes too small. For 32 KB segments, the tipping point is a T_{low} of 100 μs. For the most demanding workload of 64 KB segments, the transition is between 200–300 μs. Such large bursts make it difficult to obtain both high throughput and low delay. This is unsurprising since each segment is sent as a long series of back-to-back packets at wire speed, e.g., 64 KB is at least 40 packets.

Smoothing Bursts with Fine-Grained Pacers. The Rate Control Engine described in §3.3 introduces a pacing delay

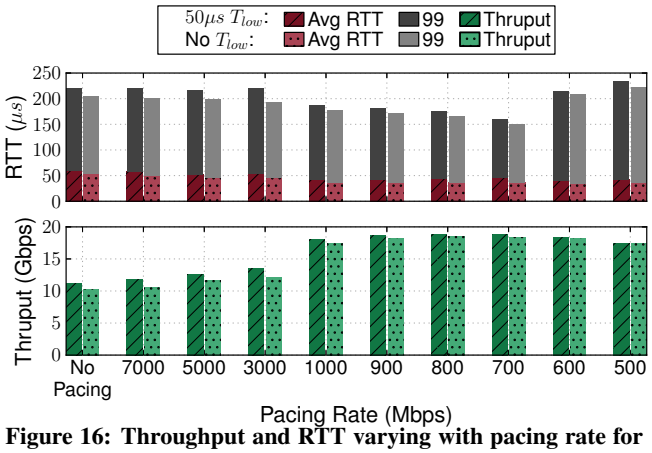

Figure 16: Throughput and RTT varying with pacing rate for 64 KB segments

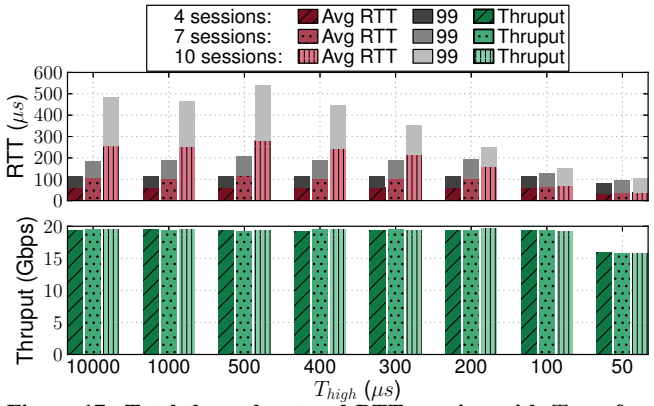

Figure 17: Total throughput and RTT varying with T_{high} for different number of connections. Segment Size = 16 KB, T_{low} = 50 μs

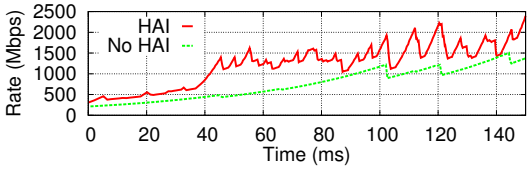

Figure 18: HAI quickly acquires available bandwidth.

between segments. To further mitigate the burstiness with large segments of 64 KB, while still enabling NIC offload, we explore fine-grained pacing. In this model, in addition to the host software pacing segments, the NIC hardware uses pacing to send the packets in each segment at a configurable rate lower than line rate. Pacing allows packets to mix more easily in the network. Programmable NICs such as NetFPGA [6] allow a pacing implementation. Prior work such as HULL [37] has also made use of NIC pacing, and fine-grained pacing queueing disciplines such as FQ/pacing are in Linux kernels [7].

We repeat the incast experiment using 64 KB segments, this time with NIC pacing, with two values of T_{low}: 0 μs and 50 μs. We are not able to implement a dynamic fine-grained pacing rate at this time and so use static rates. When computing RTTs from the completion times, we subtract the serialization delay introduced by NIC pacers to allow for comparison, e.g., pacing a 64 KB message at 1 Gbps introduces a serialization delay of 512 μs. Figure 16 shows the results for different NIC pacing rates. As expected, the reduced burstiness due to pacing leads to increase in throughput and decrease in delay, with larger throughput increases for greater pacing. The most benefit is seen at 700 Mbps: 18.9 Gbps throughput for T_{low} = 50 μs and 18.4 Gbps in the absence of any T_{low} (as opposed to 11.2 Gbps and 10.2 Gbps respectively without any pacing). Note that this also means single flow performance is capped at 700 Mbps, unless the pacing rate is adjusted dynamically. There is a slight dip in the throughput and rise in delay beyond this level, as pacing approaches the fair share and has a throttling effect.

We note that NIC hardware pacing is not an absolute requirement in TIMELY design; but rather helps navigate the tradeoff between lower network tail latency and higher CPU overhead that can be caused by software pacing with smaller segments.

Varying T_{high}. TIMELY employs a high threshold to react quickly to large RTTs. This threshold matters less than the low threshold because it only comes into play for large RTTs, but it becomes useful as the level of connection multiplexing grows and the likelihood of RTT spikes increase.

Figure 17 shows the effect on throughput as the high threshold is reduced for different numbers of competing connections. In our earlier runs with four connections per client, the 99-percentile RTTs are around 100 μs. This means that any $T_{high} > 100\mu$s has little effect. As the load climbs to 7 connections per client, the 99-percentile RTT settles close to 200 μs. Then there is a drop in RTT as we reduce T_{high} to 100 μs and below. For 10 connections per client, the 99-percentile RTTs remain close to 500 μs for T_{high} of 500 μs or more, and decrease as T_{high} falls. The throughput is quite good for T_{high} down to 100 μs, but is significantly lower at 50 μs for all three connection levels. These results show that a high threshold down to at most 200 μs helps to reduce tail latency without degrading throughput.

Hyper active increment (HAI). HAI helps to acquire available bandwidth more quickly. To show this, we perform an incast with a change in the offered load. The incast starts with 10 clients and 10 connections per client. After an initial period for convergence, every client simultaneously shuts down 9 of its 10 connections, thus increasing the fair share rate of the remaining connection by 10X. Figure 18 shows how HAI ramps up connection throughput from an initial fair rate of 200 Mbps to 1.5 Gbps within 50 ms, and reaches the new fair share of 2 Gbps in 100 ms. In contrast, a fixed additive increment only achieves 1.5 Gbps after 140 ms. We find that a HAI threshold of five successive RTTs strikes a good balance between convergence and stability.

6.2 Large-Scale Experiments

We investigate TIMELY's large-scale behavior with experiments run on a few hundred machines in a classic Clos topology [12, 34]. We show that TIMELY is able to maintain predictable and low latency in large all-to-all and incast network congestion scenarios. The experiment generates RPCs between client server pairs. To stress TIMELY and

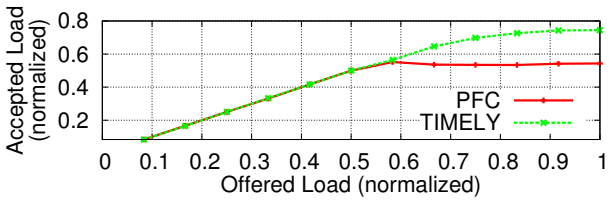

Figure 19: Accepted versus offered load (64 KB messages).

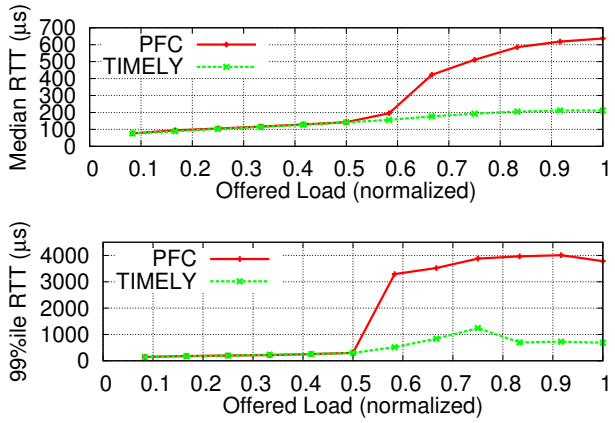

Figure 20: Median and 99-percentile RTTs as measured by pingers under different offered load.

create increased burstiness, we used 64 KB RPCs and segment sizes.

Longest path uniform random. In this traffic pattern, a client picks a server from a set of servers with the longest path through the network. Clients issue 64 KB requests. The server replies with a payload of the same size. The benchmark collects goodput and RPC latency. RPC latency is computed at the client from the time the request is sent to the server, to when it receives a response from the server.

Figure 19 shows normalized throughput (to the maximum offered load used in the experiment) as observed by the application for increasing offered loads on the x-axis. The saturation point of the network (the point at which accepted load is less than the offered load) is higher for TIMELY as it is able to send more traffic by minimizing queueing and thereby also pause frames per second.

Figure 20 shows RTT versus load. TIMELY reduces the median and 99-percentile RTT by $2X$ and $5X$ respectively compared to PFC. This results in a corresponding reduction of RPC median latency of about $2X$ (shown in Figure 21). Without TIMELY, beyond saturation the network queuing increases in an attempt to reach the offered load. With TIMELY, low network queuing is maintained by moving queuing from the shared network to the end-host (where it is included in RPC latency but not in the RTTs). Therefore, the 99-percentile of RPC latency reduction effect diminishes as the offered load increases beyond the saturation point.

Network imbalance (incast). To stress TIMELY's ability to mitigate congestion, we designed an experiment with a background load of longest path uniform random traffic and then added an incast load. We use three levels of background load: low (0.167 of the maximum offered load in Figure 19), medium (0.3) and high (0.5). Figure 22 shows the normal-

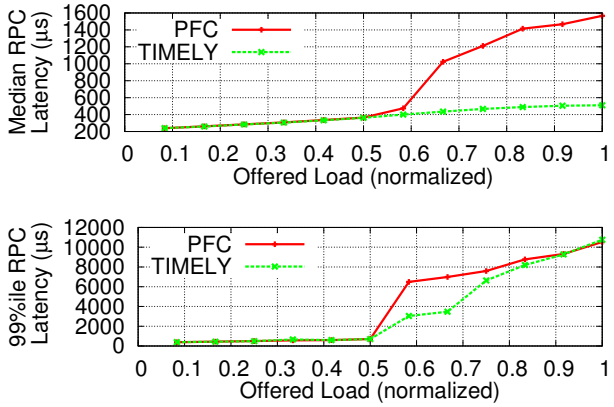

Figure 21: Median and 99-percentile RPC latencies.

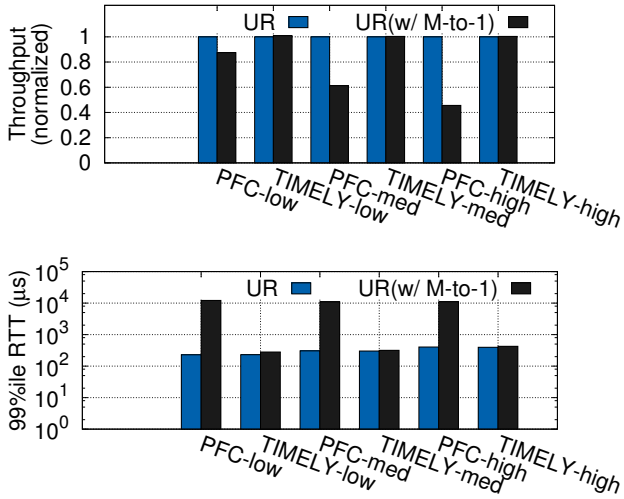

Figure 22: Adding one 40-to-1 pattern to longest path uniform random ((a) Normalized throughput (b) 99-percentile RTT).

ized throughput and 99-percentile RTT for this experiment. We normalize throughput to that of the background load. We know from Figure 19 that TIMELY and PFC throughput are the same for uniform random traffic before network saturation. When we add an incast, without TIMELY, throughput falls by 13% to 54%, depending on the background network load, primarily due to head of line blocking created by PFCs. This observation is confirmed with RTT measurements in Figure 22, which show that TIMELY is able to keep queuing relatively low, preventing congestion spreading [20, 36], by rate limiting only the flows passing along the congested path. The overall throughput for TIMELY remained the same during the incast.

Application level benchmark.

Figure 23 shows RPC latency of a datacenter storage benchmark (note that the y-axis is on a log-scale). Without TIMELY, the application is limited in the amount of data it can push through the network while keeping the 99th percentile RTT low enough. With TIMELY, the application is able to push at higher utilization levels without suffering negative latency consequences. Therefore, the drop in application data unit latency (in seconds) is really a reflection of the increased throughput that the application is able to sustain during query execution.

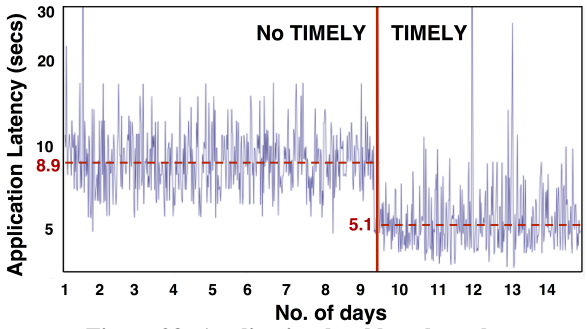

Figure 23: Application-level benchmark.

7. RELATED WORK

Datacenter congestion control is a deeply studied topic [15, 18, 35, 37, 38, 47]. TIMELY focuses on the same problem.

RED [23] and CoDel [40] drop packets early, prompting senders to reduce transmission rates to avoid the large standing queues associated with tail drops. However, loss still drives up latencies for the flows that experience packet drops. To avoid packet drops, many schemes rely on switch-support in the form of ECN, in which packets are marked to indicate congestion [22]. ECN marks are often combined across multiple packets [15, 35, 37] to provide fine-grained congestion information, but our experiments in §2 show that ECN has inherent limitations. There have also been other proposals that rely on switch support to mitigate congestion such as QCN [29] (fine-grained queue occupancy information) and pFabric [38] (fine-grained prioritization).

TIMELY belongs to a different class of algorithms that use delay measurements to detect congestion, which requires no switch-support. We take inspiration from TCP Vegas, FAST, and Compound [16, 44, 46]. These proposals are window-based and maintain a queue close to the minimum RTT. In contrast, TIMELY is a rate-based algorithm that employs a gradient approach and does not rely on measuring the minimum RTT. We show that it works well with NIC support, despite infrequent RTT signals.

A recent scheme, DX [17], independently identified the benefits of using delay as congestion signal for high throughput and low latency datacenter communications. DX implements accurate latency measurements using a DPDK driver for the NIC and the congestion control algorithm is within the Linux TCP stack. DX algorithm is similar to the conventional window-based proposals, with an additive increase and a multiplicative decrease that's proportional to the average queuing delay.

CAIA Delay Gradient [25] (CDG) proposes a delay gradient algorithm for TCP congestion control for wide-area networks. Its key goal is to figure out co-existence with loss based congestion control. Hence the nature of its algorithms are different from those in TIMELY.

Link-layer flow control is used for low-latency messaging in Infiniband and Data Center Bridging (DCB) networks. However, problems with Priority Flow Control (PFC), including head of line blocking and pause propagation or congestion spreading, are documented in literature [20, 36]. Some recent proposals aim to overcome these issues with PFC using ECN markings to maintain low queue occupancy. TCP-Bolt [14] uses modified DCTCP algorithm within the kernel TCP stack. DCQCN [48] uses a combination of ECN markings with a QCN-inspired rate-based congestion control algorithm implemented in the NIC. Evaluations demonstrate that it addresses the HoL blocking and unfairness problems with PFC, thus making RoCE viable for large-scale deployment. TIMELY uses RTT signal, is implemented in host software with support of NIC timestamping, and is applicable to both OS-bypass and OS-based transports. Comparison of TIMELY and DCQCN in terms of both congestion control and CPU utilization is an interesting future work.

Congestion can also be avoided by scheduling transmissions using a distributed approach [18, 47] or even a centralized one [30]. However, such schemes are yet to be proven at scale, and are more complex than a simple delay-based approach.

Finally, load-sensitive routing such as Conga [13] and FlowBender [11] can mitigate congestion hotspots by spreading traffic around the network, thereby increasing throughput. However, host-based congestion control is still required to match offered load to the network capacity.

8. CONCLUSION

Conventional wisdom considers delay to be an untrustworthy congestion signal in datacenters. Our experience with TIMELY shows the opposite – when delay is properly adapted, RTT strongly correlates with queue buildups in the network. We built TIMELY, which takes advantage of modern NIC support for timestamps and fast ACK turnaround to perform congestion control based on precise RTT measurements. We found TIMELY can detect and respond to tens of microseconds of queueing to deliver low packet latency and high throughput, even in the presence of infrequent RTT signals and NIC offload. As datacenter speeds scale up by an order of magnitude, future work should focus on how effective RTTs continue to be for congestion control, alongside rethinking the nature of delay based algorithms.

9. ACKNOWLEDGMENTS

We thank Bob Felderman, Ashish Naik and Philip Wells whose participation and feedback made the work and this submission possible. We thank Mike Marty, Joel Scherpelz, and Dan Gibson for their direct contributions to this work; Sridhar Raman, and Denis Pankratov for their contributions to NIC pacing and hardware timestamps; Mike Ryan for application benchmarks; Varun Gupta and Abdul Kabbani for their insights on various congestion signals; Laurent Chavey and Bill Berryman for their ongoing support of congestion control work. We thank our shepherd, Mohammad Alizadeh, and the anonymous reviewers for providing excellent feedback.

10. REFERENCES

[1] Chelsio T5 Packet Rate Performance Report. http://goo.gl/3jJL6p, Pg 2.

[2] Data Center Bridging Task Group. http://www.ieee802.org/1/pages/dcbridges.html.

[3] Dual Port 10 Gigabit Server Adapter with Precision Time Stamping. http://goo.gl/VtL5oO.

[4] Gnuplot documentation. http://goo.gl/4sgrUU, Pg. 48.

[5] Mellanox for Linux. http://goo.gl/u44Xea.

[6] The NetFPGA Project. http://netfpga.org/.

[7] TSO Sizing and the FQ Scheduler. http://lwn.net/Articles/564978/.

[8] Using Hardware Timestamps with PF RING. http://goo.gl/oJtHCe, 2011.

[9] Who (Really) Needs Sub-microsecond Packet Timestamps? http://goo.gl/TI3r1u, 2013.

[10] A. Kabbani et al. AF-QCN: Approximate Fairness with Quantized Congestion Notification for Multi tenanted Data Centers. In *Hot Interconnects'10*.

[11] A. Kabbani et al. FlowBender: Flow-level Adaptive Routing for Improved Latency and Throughput in Datacenter Networks. In *ACM CoNEXT '14*.

[12] A. Singh et al. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter. In *SIGCOMM'15*.

[13] Alizadeh et al. CONGA: Distributed Congestion aware Load Balancing for Datacenters. In *SIGCOMM '14*.

[14] B. Stephens et al. Practical DCB for improved data center networks. In *Infocom 2014*.

[15] B. Vamanan et al. Deadline-aware datacenter TCP (D2TCP). In *SIGCOMM '12*.

[16] Brakmo et al. TCP Vegas: new techniques for congestion detection and avoidance. In *SIGCOMM '94*.

[17] C. Lee et al. Accurate Latency-based Congestion Feedback for Datacenters. In *USENIX ATC 15*.

[18] C.-Y. Hong et al. Finishing Flows Quickly with Preemptive Scheduling. In *SIGCOMM '12*.

[19] D.-M. Chiu and R. Jain. Analysis of the Increase and Decrease Algorithms for Congestion Avoidance in Computer Networks. *Comput. Netw. ISDN Syst.*, 1989.

[20] D. Zats et al. DeTail: Reducing the Flow Completion Time Tail in Datacenter Networks. In *SIGCOMM '12*.

[21] J. Dean and L. A. Barroso. The Tail at Scale. *Communications of the ACM*, 56:74–80, 2013.

[22] S. Floyd. TCP and explicit congestion notification. *ACM SIGCOMM CCR*, 24(5), 1994.

[23] S. Floyd and V. Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Trans. Netw.*, 1, August 1993.

[24] I. Grigorik. Optimizing the Critical Rendering Path. http://goo.gl/DvFfGo, Velocity Conference 2013.

[25] D. A. Hayes and G. Armitage. Revisiting TCP Congestion Control using Delay Gradients. In *Networking IFIP*, 2011.

[26] D. A. Hayes and D. Ros. Delay-based Congestion Control for Low Latency.

[27] C. Hollot, V. Misra, D. Towsley, and W.-B. Gong. A control theoretic analysis of RED. In *IEEE Infocom '01*.

[28] C. Hollot, V. Misra, D. Towsley, and W.-B. Gong. On designing improved controllers for AQM routers supporting TCP flows. In *IEEE Infocom '01*.

[29] IEEE. 802.1Qau - Congestion Notification. http://www.ieee802.org/1/pages/802.1au.html.

[30] J. Perry et al. Fastpass: A Centralized "Zero-Queue" Datacenter Network. In *SIGCOMM '14*.

[31] R. Jain, D. Chiu, and W. Hawe. A Quantitative Measure of Fairness and Discrimination for Resource Allocation in Shared Computer Systems. In *DEC Research Report TR-301*, 1984.

[32] D. Katabi, M. Handley, and C. Rohrs. Internet Congestion Control for Future High Bandwidth-Delay Product Environments. In *SIGCOMM'02*.

[33] F. P. Kelly, G. Raina, and T. Voice. Stability and fairness of explicit congestion control with small buffers. *Computer Communication Review*, 2008.

[34] M. Al-Fares et al. A Scalable, Commodity Data Center Network Architecture. SIGCOMM '08.

[35] M. Alizadeh et al. Data center TCP (DCTCP). In *SIGCOMM '10*.

[36] M. Alizadeh et al. Data Center Transport Mechanisms: Congestion Control Theory and IEEE Standardization. In *Annual Allerton Conference '08*.

[37] M. Alizadeh et al. Less Is More: Trading a Little Bandwidth for Ultra-Low Latency in the Data Center. In *NSDI '12*.

[38] M. Alizadeh et al. Deconstructing datacenter packet transport. In *ACM HotNets*, 2012.

[39] N. Dukkipati et al. Processor Sharing Flows in the Internet. In *IWQoS*, 2005.

[40] K. Nichols and V. Jacobson. Controlling queue delay. *Queue*, 10(5):20:20–20:34, May 2012.

[41] J. Postel. Transmission Control Protocol. RFC 793, 1981. Updated by RFCs 1122, 3168, 6093, 6528.

[42] S. Ha et al. CUBIC: A New TCP-Friendly High-Speed TCP Variant. *SIGOPS Operating System Review '08*.

[43] S. Radhakrishnan et al. SENIC: scalable NIC for end-host rate limiting. In *NSDI 2014*.

[44] K. Tan and J. Song. A compound TCP approach for high-speed and long distance networks. In *IEEE INFOCOM '06*.

[45] V. Vasudevan et al. Safe and effective fine-grained TCP retransmissions for datacenter communication. In *SIGCOMM '09*.

[46] D. X. Wei, C. Jin, S. H. Low, and S. Hegde. FAST TCP: Motivation, Architecture, Algorithms, Performance. *IEEE/ACM Trans. Netw.*, 2006.

[47] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowtron. Better never than late: meeting deadlines in datacenter networks. In *SIGCOMM '11*.

[48] Y. Zhu et al. Congestion Control for Large-Scale RDMA Deployments. In *SIGCOMM 2015*.