TCP Inigo:

Ambidextrous Congestion Control

LA-UR-15-27631

Andrew G. Shewmaker shewa@lanl.gov [*], Carlos Maltzahn [**], Katia Obraczka [**], and Scott Brandt [**]

[*]Los Alamos National Laboratory and UC Santa Cruz
[**]UC Santa Cruz

Abstract

Some TCP congestion control algorithms [1, 34] can provide impressive performance with the help of Explicit Congestion Notification (ECN) or improved timestamping. However, configuring switch queues to mark ECN appropriately or separating low latency protocols from aggressive legacy traffic is not always feasible. And the inertia and variety of networks makes modifying both senders and receivers, altering drivers, or adding new TCP options a daunting challenge.

We propose TCP Inigo, a delay-based congestion control variant of TCP that has low barriers to deploy and can coexist with loss-based TCPs. Inigo adds two independent types of congestion control: a Round-Trip-Time (RTT) based congestion ratio on the sender, and a Relative Forward Delay (RFD) based congestion ratio used by the receiver to either mark ECN or manage the receive window. We compare and contrast these uses of RTT and RFD with ECN, demonstrating more than 2× smaller RTTs than non-ECN TCPs and superior utilization in easily reproduced Mininet experiments.

1 Introduction

Congestion control remains a perennial concern in data centers and the Internet. Network congestion is one of the causes of variation in delay and can also cause performance collapse in a worst case scenario. For businesses the long tail of variations can cost money [17], but congestion made worse by bufferbloat [22] creates pain every day for home users.

While many advances in congestion control [1, 34] and active queue management (AQM) [39, 47, 37] show potential, almost all proposals require amounts of change and configuration that prevent the new technologies from spreading quickly. A brief overview of DCTCP, which provided inspiration for Inigo, and some of its complications are in § 2.1. There is considerable practical value in being able to improve network performance while minimizing the effort needed to deploy and maintain the changes. Any work that must be redone for unique hardware or that increases the complexity of network configurations will likely find itself in a losing tug-of-war with End-to-end arguments [43, 4].

There has been a long line of TCP congestion control variants that try to keep congestion low and do not require extensive change or configuration of network equipment, but most are unable to compete for bandwidth with loss-based TCPs [7]. Some, such as CAIA Delay-Gradient (CDG) TCP [26], are able to coexist to some degree. Others, like Performance-oriented Congestion Control (PCC) [18] may have a promising prototype, but are still far from deployable.

With the difficulty of changing both Local Area and the Internet in mind, this paper proposes TCP Inigo, which offers the following contributions that do not require any switch configuration, adapter support, driver modification, or new TCP options:

- a new fallback mode for DCTCP
  - mimics DCTCP's ECN behavior using RTTs
  - shares bandwidth with loss-based TCPs
  - attains best-in-class latencies
  - requires only a sender-side modification
  - available as Linux kernel module
- receiver congestion control *or* ECN marking
  - mimics DCTCP's ECN behavior using OWDs
  - DCTCP-style receive window [1] for *all* TCPs
  - requires only a receiver-side modification
  - available as Linux kernel patch

[1]While shrinking the receive window is strongly discouraged, we can show that Inigo's small adjustments result in fewer resent segments, fairer bandwidth sharing, and lower latencies.

The rest of this paper is organized as follows:

Section § 2 describes the Inigo sender-side modification, along with the complications surrounding existing TCP RTT measurements and how they can be used. Section § 3 describes the Inigo receiver-side modification, and how the differences in OWDs can drive similar congestion control decisions with regard to ECN marking and window sizing. Section § 4 demonstrates the effectiveness of both techniques independently and combined. Next, Section § 5 describes related work and the conclusion § 6. In addition, the availability of the TCP Inigo code and experiments is detailed in Appendix A, followed by supplementary material.

2 TCP Inigo Sender

TCP Inigo is composed of two independent techniques. The first is a sender-side only modification that uses TCP RTT measurements. Both follow in the footsteps of DCTCP in using a congestion ratio, a measure of the extent of congestion, in order to proportionally adjust the congestion window.

2.1 ECN and DCTCP

Standard ECN support [42] directs a sender to halve its window once per RTT upon seeing an marked with Congestion Exists (CE), while DCTCP [1] tracks the ratio of bytes marked with CE to the total number of bytes acknowledged (ACKed) in order to estimate the extent of congestion. Given congestion markings on all of the packets in a window, DCTCP will halve the window. If DCTCP sees fewer markings it will back off proportionally less.

When ECN is not supported by the receiver, DCTCP falls back to basic TCP Reno. If the receiver supports ECN, but was not modified to accurately convey ECN with delayed ACKs, then DCTCP will under-estimate the extent of congestion. Kato developed a one-sided variant of DCTCP [30], but it compromises the performance of DCTCP when the receiver has been modified appropriately.

Switches must also be configured to mark ECN appropriately for use with DCTCP. Configuring for DCTCP is simpler than for Random Early Detection (RED) [21], although it can use the same widespread support in Ethernet hardware. However, there are many situations where DCTCP cannot be easily deployed. A cloud provider may not be able to force all tenants to use a buffer-friendly TCP, configuring separate switch queues may be considered impractical, or setting per-route congestion control on the application side may not be fine-grained enough.

In cases like those, it would be good to fall back to behavior as similar as possible, but with fewer requirements. We submit that existing RTT measurements are adequate to the task. Up until this point, using existing RTTs has not resulted in low queue depths and tight latency distributions similar to DCTCP.

2.2 TCP RTTs

TCP's timestamps and RTT measurements are taken several layers and queues above the hardware. As such, they include the variability of the host operating systems and not just the network delay due to congestion. Recently, Lee, et al. proposed DX congestion control which can utilize improved timestamping to good effect [34]. However, those improvements rely on driver modifications, new TCP header options, and changes to both senders and receivers. Those changes will require a determined effort to become common enough to rely on.

In addition to ease of deployment, there are other advantages to using already available time measurements for congestion control. If the goal is to minimize the end-to-end delay variability up to the application layer, then the fact that the TCP RTT includes delays due to OS buffers and network buffers may be an asset. Also, a holistic observation like the RTT does not combine independent signals in a way that might indicate more congestion than is actually present. In contrast, ECN marking at switches is done independently, so one could envision an unlikely scenario where a series of switches each experienced minor congestion at different times, causing the majority of a flow's packets to be marked.

RTT measurements are noisy, so reacting to individual measurements results in unpredictable behavior. That is why many algorithms use some sort of smoothing, but given the dynamic range of RTTs this can often prevent quick responses to changing conditions.

DCTCP-inspired delay-based congestion control was made more effective by several developments in the Linux kernel since the original DCTCP paper. Internal buffer bloat and the delay variability that comes with it were much improved with features such as Byte Queue Limits [12], TCP Small Queues [14], and TCP Segmentation Offload (TSO) sizing and pacing [15]. In addition, the units of the sender's RTT measurement changed from milliseconds to microseconds [19].

2.3 RTT Congestion Ratio

Inigo uses *late RTTs* in the same way that DCTCP uses ECN markings to calculate and respond to the extent of congestion. Since the RTT signal arrives at the same frequency as ECN markings (i.e. every ACK), and since TCP RTTs must generally correspond to increased queuing (despite noise), we believe that the recommended

DCTCP threshold is valid for defining a what makes an RTT late.

Alizadeh, et al. derived equation (1), in which C and d respectively denote the bottleneck capacity (packets/sec) and propagation delay (in sec), giving a threshold of K (packets). This threshold is 2.7% larger than their original, and is based on a fluid model of DCTCP that is more accurate than their previous sawtooth model [2].

$$K \approx 0.17Cd \tag{1}$$

The corresponding queuing delay threshold, d_{thresh}, is simply K divided by the bottleneck capacity C. Substituting equation (1) into (2) gives (3).

$$d_{thresh} = K/C \tag{2}$$

$$d_{thresh} \approx 0.17Cd/C = 0.17d \tag{3}$$

In their fluid model, the RTT at time t of a flow is given by equation (4), where d is the propagation delay, $q(t)$ is the queue size at the switch and $q(t)/C$ is the queuing delay. But since d is not doubled as would be expected for one-way propagation delay, d is actually the minimum RTT, and therefore d_{thresh} is the RTT threshold.

$$R(t) = d + q(t)/C \tag{4}$$

Algorithm 2 calculates the congestion ratio α_{RTT} using the RTTs marked late by algorithm 1. It is nearly identical to the approach taken in DCTCP, where a fraction F is tracked during a window and used to update the exponential weighted moving average of the congestion ratio α_{RTT}.

Algorithm 1 RTT Congestion Marking

for each ACK **do**
  if $RTT_{min} = 0 \lor RTT < RTT_{min}$ **then**
    $RTT_{min} \leftarrow RTT$
  end if
  $RTTs_{observed} \leftarrow RTTs_{observed} + 1$
  if $RTT \geq RTT_{min} + d_{thresh}$ **then**
    $RTTs_{late} \leftarrow RTTs_{late} + 1$
  end if
end for

Algorithm 2 Congestion Ratio with RTTs

for every window **do**
  $F \leftarrow RTTs_{late}/RTTs_{observed}$
  $\alpha_{RTT} \leftarrow (1-g) \times \alpha_{RTT} + g \times F$
  $RTTs_{observed} \leftarrow 0$
  $RTTs_{late} \leftarrow 0$
end for

Our delay-based congestion ratio uses RTT observations instead of bytes, and delayed ACKs are not compensated for since the receiver can only improve the signal by sending ACKs more frequently. The number of RTT measurements can be further reduced due to TSO. Attempts to reason about best and worst case congestion scenarios given a RTT measurement and the number of segments being aggregated by TSO resulted in worse performance, but we describe what was attempted in appendix B.

Algorithms 1 and 2 work well in simple homogeneous environments, but Inigo will have to co-exist with other traffic that does not keep switch buffer occupancy low. However, that situation can be addressed by dilating RTT_{min} as in algorithm 3. Dilation also allows Inigo to adapt to truly increasing RTTs: new longer route, changes to traffic shaping, or overheads from potentially adaptive lower level changes like Forward Error Correction for link degradation.

If major congestion persists for several RTTs, then α_{RTT} will approach its maximum and Inigo will approximately halve its window multiple times, as explained in § 2.5. But in the presence of overly aggressive flows, α_{RTT} will not decrease as it should. In that situation Inigo responds aggressively by changing its late RTT marking threshold to be the higher average RTT, resetting its measure of congestion to zero, and entering Slow Start. Inigo eases off after a few windows so that the queue can drain in the case conditions have gotten better again.

Algorithm 3 RTT_{min} Dilation

$dilate \leftarrow 0$
for every window **do**
  if $\alpha_{RTT} > 9\alpha_{max}/10 \lor dilate > 3$ **then**
    $dilate \leftarrow dilate + 1$
  end if
  if $3 < dilate < 7$ **then**
    $RTT_{predilate} \leftarrow RTT_{min}$
    $RTT_{min} \leftarrow RTT_{smoothed} + d_{thresh}$
    $\alpha_{RTT} \leftarrow 0$
    $ssthresh \leftarrow ssthresh_{max}$
  end if
  if $dilate > 7$ **then**
    $RTT_{min} \leftarrow RTT_{predilate} + d_{thresh}$
    $dilate \leftarrow 0$
  end if
end for

2.4 Slow Start

There are many variations of TCP Slow Start, in which the initial rate of transmission rapidly increases, usually via window doubling. This phase is necessary because TCP does not know the speed of the network. DCTCP

shows that the standard method of doubling the window size every RTT can quickly achieve full throughput while keeping queue occupancy low with the help of ECN marking on switches. It exits Slow Start immediately upon seeing an ACK with CE. Matching that behavior using only delay is slightly more challenging, with HyStart [23] probably the most successful example of the technique.

The Linux kernel possesses implementations of HyStart in TCP CUBIC [24] and CDG [26], with the latter containing some experimental changes. Both variants detect congestion during Slow Start with ACK trains and when a late RTT is observed. They take the minimum of the first seven RTT samples and exit Slow Start immediately upon receiving one late RTT. The delay threshold, d_{thresh}, used by CUBIC in Linux 4.2.0 is one eighth the minimum RTT, bounded between 32 and 128 milliseconds. CDG's d_{thresh} is also one eighth the minimum RTT, but it is calculated with a microsecond clock and only has an upper bound of 125 microseconds.

HyStart was designed to find an early, safe exit point to enter CUBIC's aggressive Congestion Avoidance phase. But the threshold was increased to one eighth in 2014 because one sixteenth was too sensitive. That oversensitivity was one of the reasons Linux networking maintainer David Miller recommended disabling HyStart by default [35]. Interestingly, CUBIC's new threshold is within 1.8% of the initially recommended threshold for DCTCP [1].

Inigo sets aside HyStart's ACK train heuristic, exiting Slow Start only upon seeing an RTT that exceeds $RTT_{min} + d_{thresh}$, as in algorithm 2. Similar to HyStart, Inigo requires a minimum number of observations to initialize RTT_{min}. But instead of simply exiting Slow Start by setting the Slow Start threshold *ssthresh* to the current congestion window *cwnd*, Inigo uses the congestion ratio to decrease the congestion window. Algorithm 4 uses $RTTs_{observed}$ and $RTTs_{late}$ from algorithm 2. If the congestion ratio is non-zero once 10 RTTs are observed, then it reduces *cwnd* by the congestion ratio similarly to algorithm 7 in § 2.5. Then it assigns $ssthresh = cwnd$.

Algorithm 4 Slow Start

```
for every ACK do
    if cwnd ≤ ssthresh ∧ samples ≥ 10 then
        F ← RTTs_late / RTTs_observed
        α_RTT ← (1 − g) × α_RTT + g × F
        if α_RTT > 0 then
            ssthresh ← cwnd − cwnd_cnt × α/2
        end if
    end if
end for
```

2.5 Congestion Avoidance and Response

The basic congestion avoidance logic first implemented by TCP Inigo is shown in algorithm 5. Other than taking the maximum of α_{RTT} from algorithm 2 and α_{DCTCP}, it works exactly the same as DCTCP and reduces the congestion window, *cwnd*, in proportion to the congestion ratio. Taking the maximum congestion ratio should allow accurate congestion control to function across connections that include and assortment of switches with and without DCTCP-style ECN marking.

Algorithm 5 Congestion Avoidance and Response

```
for every window do
    α ← max(α_RTT, α_DCTCP)
    cwnd ← cwnd × (1 − α/2)
    if α > 0 then
        cwnd ← cwnd + 1
    end if
end for
```

However, Alizadeh, et al. proposed an RTT-fairness enhancement to DCTCP [2], in which it would respond to congestion upon every ACK. The improvement counteracts the typical TCP behavior of flows with longer RTTs growing more slowly than flows with short RTTS by causing the latter to respond to congestion more rapidly. Conventional wisdom is for a congestion response to only occur once per RTT in order to see the effect of the response, but the sum of the adjustments of the per-ACK response are designed to approximate the normal response once per RTT.

Passengers in a vehicle appreciate a driver who makes micro-adjustments instead of slamming on the breaks or the accelerator, even if the average speed is the same. Similarly, a TCP that makes sub-window adjustment should be able to avoid over-steering. This is about more than RTT-fairness. It affects convergence time for long lived flows with the same RTT starting at different times. Flows beginning earlier will have a larger window and respond more slowly than newer flows. And sub-window adjustments should allow mice flows to enter and leave with smaller perturbations to elephant flows.

Unfortunately, the units of the Linux `snd_cwnd` variable are in packets and the DCTCP RTT-fairness update algorithm 6 implies the need to adjust the window by a fraction of a packet. For example, if $W = 200$ and $\alpha = 300/1024$, then upon seeing and ECN marking $W \leftarrow 100 + 1/200 - 150/1024 \approx 199.86$. Integer arithmetic would result in $W \leftarrow 200$, and if α remains relatively constant, then $W \leftarrow 200$ after a window of ACKs. On the other hand, the original once per RTT response would yield $W \leftarrow 171$.

The sender's window could be tracked in bytes like

Algorithm 6 DCTCP RTT-fairness

for every ACK **do**

$$W \leftarrow W + \begin{cases} 1/W & \text{if ECN } = 0 \\ 1/W - \alpha/2 & \text{if ECN } = 1 \end{cases}$$

end for

the receiver's window, allowing fine-grained changes to accumulate. Or the window mechanism might be altered to allow sending at a different frequency, as has been proposed for scaling to small RTTs [8]. However, both would require either modifying the whole TCP stack or adding extra variables to the TCP congestion structure for private per-socket data. A sub-window approach to RTT-fairness is simpler to implement and requires fewer variables.

Linux implements $W \leftarrow W + 1/W$ (i.e. Congestion Avoidance) with a counter `snd_cwnd_cnt`, which is incremented by the number of ACKed packets until `snd_cwnd_cnt` reaches `snd_cwnd`, and `snd_cwnd` is incremented by one. Similarly `snd_cwnd` can be decremented by a fractional packet by responding every N ACKs as in algorithm 7. In our experiments, we observed that a sub-window response sometimes backs off a little too much, and we found that Congestion Avoidance of $W \leftarrow W + 2/W$ ensured better performance.

Algorithm 7 Sub-window Congestion Response

for every W_{sub} ACKs, where $1 < W_{sub} < W$ **do**
  if $\alpha > 0$ **then**
    $W \leftarrow W - W_{sub}\alpha/2$
  end if
end for
for every window **do**
  $cwnd \leftarrow cwnd + 2$
end for

The frequency of response must be balanced with sensitivity to the congestion ratio, α_{min}, calculated with equation (5). For instance, a response interval $W_{sub} = 20$ and a maximum congestion ratio $\alpha_{max} = 1024$ would be able to adjust the window in response to $\alpha \geq 104$. The smallest sub-window that could make any adjustments below $\alpha = \alpha_{max}$ would be $W_{sub} = 3$, with $\alpha \geq 684$. Note that increasing the scaling factor of α does not significantly improve sensitivity. And a certain amount of insensitivity to α is beneficial because it prevents responding to occasional large RTTs caused by operating system noise.

$$\alpha_{min} \leftarrow \lceil 2\alpha_{max}/W_{sub} \rceil + \begin{cases} 1 & \text{if} \alpha_{min} \bmod 2 = 1 \\ 0 & \text{if} \alpha_{min} \bmod 2 = 0 \end{cases} \quad (5)$$

In our experiments, we have found that algorithm 5 occasionally results in the fairest bandwidth sharing, but algorithm 7 can achieve a much tighter distribution of bottleneck queue lengths and delay with good bandwidth fairness.

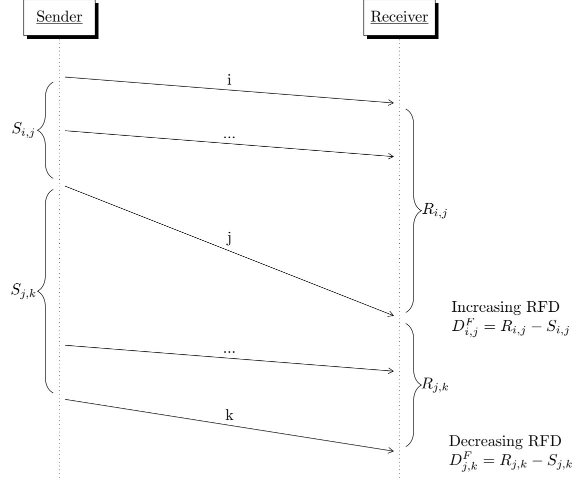


Figure 1: Examples of Increasing and Decreasing Relative Forward Delay Measurements

3 TCP Inigo Receiver

The second, separate congestion control that makes up TCP Inigo is a receiver-side only modification that detects congestion by monitoring the accumulation of differences in One Way Delays (OWDs). If the sender is an ECN-Capable Transport, then the receiver marks Congestion Encountered (CE) in the header of the next ACK. Otherwise, the receiver controls congestion in a style similar to DCTCP via the receive window.

3.1 Using Relative Forward Delay

Relative Forward Delay (RFD) was defined as the difference of OWDs by Parsa, et al. [40] when they used it in the congestion control of TCP Santa Cruz. Example RFDs are shown in Figure 1, where S is the delta between send timestamps, R is the difference between receive timestamps, and RFD is D^F, the delta between any pair of S and R. Calculating RFD does not require clock synchronization, but it does require stable clocks of the same resolution. Appendix C contains more background and a short discussion of timestamp resolution.

Algorithm 8 shows how the running RFD total of D^F_{total} and d_{thresh} based on RTT_{min} can be used to mark bytes as late, similarly to DCTCP with ECN and Inigo's sender with RTTs.

The receiver keeps track of the running sum of RFD.

Algorithm 8 RFD Congestion Marking

```
for each packet received do
    if RTT_min = 0 ∨ RTT < RTT_min then
        RTT_min ← RTT
    end if
    bytes_total ← bytes_pkt
    S_{i,j} ← tsval_j − tsval_i
    R_{i,j} ← tsecr_j − tsecr_i
    if S_{i,j} = 0 ∧ R_{i,j} = 0 then return
    end if                  ▷ clock resolution low, keep tsX_i
    D^F_{i,j} ← R_{i,j} − S_{i,j}
    D^F_total ← max(0, D^F_total + D^F_{i,j})
    if D^F_total ≥ d_thresh then
        bytes_late ← bytes_pkt
        ECN ← 1
    end if
    tsval_i ← tsval_j
    tsecr_i ← tsecr_j
end for
```

If the total RFD becomes negative, then that means the measurements started taking place when congestion was already in progress, and therefore the total RFD is zeroed as a new baseline. Whenever the total RFD exceeds d_{thresh} given by equation (3), the receiver marks Congestion Encountered (CE) bits on the next ACK. This is done just before the code that DCTCP added to accurately transmit the extent of congestion using ECN despite delayed ACKs.

Algorithm 9 Congestion Ratio with RFDs

```
for every window do
    F ← bytes_late / bytes_total
    α_RFD ← (1 − g) × α_RFD + g × F
    bytes_total ← 0
    bytes_late ← 0
end for
```

In the case where the sender does not support ECN, the receiver tracks the congestion ratio and modifies the receive window in a fashion similar to algorithms 4 and 5, except in bytes and with an immediate ACK sent every congestion window change. Note that Slow Start exits when D^F_{total} exceeds the d_{thresh} since it the millisecond clock requires many packets before RFD can be measured.

RFCs 793 and 1122 strongly discourage shrinking the receive window since in-flight packets would be dropped, but they also say that senders should be prepared for that case [41, 6]. However, Inigo makes relatively small DCTCP-style adjustments, and frequently updates the sender when sub-window congestion responses are enabled. Therefore, only the minimum number of packets necessary to prevent congestion are in danger of being dropped. And Linux, at least, does not appear to be in danger of dropping packets due to a shrinking receive window. It keeps quadruple the amount copied to user space in the last RTT in order to handle packet losses, Slow Start, and three RTTs worth of data. Experiments with Inigo show that shrinking the receive window carefully results in fewer retransmitted segments than would normally occur.

TCPs that wish to implement receiver-side congestion control like Inigo should ensure that their receive buffer is at least twice the size of the congestion window. This will prevent in-flight packets from being dropped during extreme congestion when the window is halved over the span of one RTT.

Theoretical benefits of the receiver marking ECN or controlling the sender's window include:

- no switch configuration is necessary to use ECN
- it forces all TCPs sending to the receiver to keep buffer occupancy low
- window is controlled in bytes instead of packets (finer granularity)
- robust with regard to ACK loss (sender immediately knows RFD-based measure of congestion upon next ACK, no miscalculation due to missing information)

4 Experiments

We have created TCP Inigo, which has two independent types of congestion control. The key feature of the first component of Inigo is an RTT-based congestion ratio on the sender, used as a fallback for DCTCP instead of TCP Reno when ECN is not supported. The key feature of the second component is an RFD-based congestion ratio used by the receiver to either mark ECN or control the receive window.

Next we present Mininet [25] convergence and incast experiments based on those found commonly found in papers since DCTCP [1]. In all cases, iperf2 [48] clients generate the flows to one server. Mininet is configured to provide a simple star topology with links running at 500Mbps and a 2 millisecond one way delay between hosts.

Mininet does not currently allow link bandwidths above 1Gbps, and the fidelity of experiments can suffer long before that, depending on the system. However, Mininet does allow rapid development and easy reproduction of experiments. These results are preliminary. Inigo will need to be tested with a much greater variety of conditions before it is ready for production use.

For experiments using iperf2, bandwidth is shown with small stacked bar graphs, with time on the X axis and identical Y scales. The specific values are less important than the ability to see at a glance whether or not the expected pattern of flow bandwidths has been achieved. In addition to the stacked bandwidth graphs, we include graphs of the empirical Cumulative Distribution Functions of the switch's bottleneck queue and the TCP sender's Smoothed RTT (SRTT).

The TCP variants tested in the experiments include CDG (a state of the art delay-based TCP that does not require special support), CUBIC (an aggressive loss-based TCP that is default in many Linux distributions), DCTCP (a state of the art ECN-based TCP), Inigo, and Reno (DCTCP's fallback). If the network is configured to mark ECN, the TCP's name is suffixed with *+ECN*. Likewise, when a TCP is sending to Inigo's receiver-side congestion control or ECN marking, its name is suffixed with *RCV_CC* or *RCV_ECN*, respectively.

4.1 Five Flows

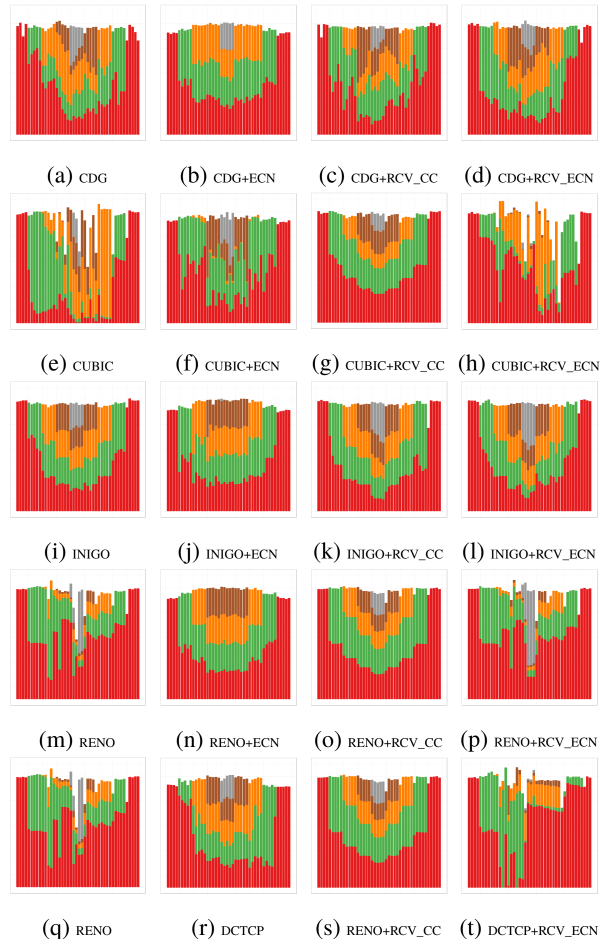
In this experiment each iperf2 client precedes the next by five seconds and continues transmitting five seconds longer than the client that follows it. Fair sharing should look somewhat like the cross-section of stacked bowls, with stair-steps at five second intervals.

Figure 2 shows a wide variety of bandwidth patterns, with consistently high utilization, crisper stair-steps, and fairer sharing in row 2i and column 2c, corresponding to Inigo's sender and receiver-side congestion control, respectively. Unfortunatley, Inigo's receiver-side ECN marking does not appear to work well. Since the receiver's congestion control does work well and they share almost all the same code, we conclude that our current implementation is faulty. Also, Inigo+ECN 2j shows that there is some negative interaction the combined DCTCP and Inigo congestion control.

CDG's graphs, row 2a, exhibit the correct basic pattern, but the stair-steps are only clear in 2b when the network is configured with ECN. CUBIC, row 2e, suffers from particularly unfair bandwidth sharing except in the case that it is talking to an Inigo receiver 2h. Even ECN does not help CUBIC as much as receiver-side congestion control.

Reno 2m looks slightly better than CUBIC since it does not overflow the switch's buffer as frequently. DCTCP has only two unique cases compared to Reno. The case it was designed for is depicted in Figure 2r, which does not look quite as good as expected since the second flow appears to have a difficult time getting started.

Of course, good utilization and fair bandwidth sharing is only part of the story. A link can be kept fully utilized if its buffer is kept filled to capacity, but the question



(a) CDG (b) CDG+ECN (c) CDG+RCV_CC (d) CDG+RCV_ECN
(e) CUBIC (f) CUBIC+ECN (g) CUBIC+RCV_CC (h) CUBIC+RCV_ECN
(i) INIGO (j) INIGO+ECN (k) INIGO+RCV_CC (l) INIGO+RCV_ECN
(m) RENO (n) RENO+ECN (o) RENO+RCV_CC (p) RENO+RCV_ECN
(q) RENO (r) DCTCP (s) RENO+RCV_CC (t) DCTCP+RCV_ECN

Figure 2: Five Flows Converge - Bandwidth
Consistently crisper stair-steps and fairer sharing in row 2i and column 2c, corresponding to Inigo's sender and receiver-side congestion control

is: how low a buffer can be kept without letting it drain completely too often? Figure 3 shows how well various TCPs do with and without an Inigo Receiver to help. An Inigo sender keeps the shortest queues when ECN is unavailable, and Inigo receiver-side congestion control helps CUBIC and Reno reduce their queues by almost $4\times$.

It can be seen in Figure 5 that the Inigo sender approaches the low queue length of ECN-enabled TCPs—only exceeding 20 packets approximately 10% of the time. Inigo's receiver-side congestion control does not appear to help further lower the queue depth attained by the Inigo sender. Figure 4 confirms our suspicion that our implementation of Inigo's receiver-side ECN marking is flawed, as it is having little to no effect on any sender. Since that it is the case, the corresponding CDFs will be excluded in the incast experiment below.

Figures 6 and 7 show a high correlation between

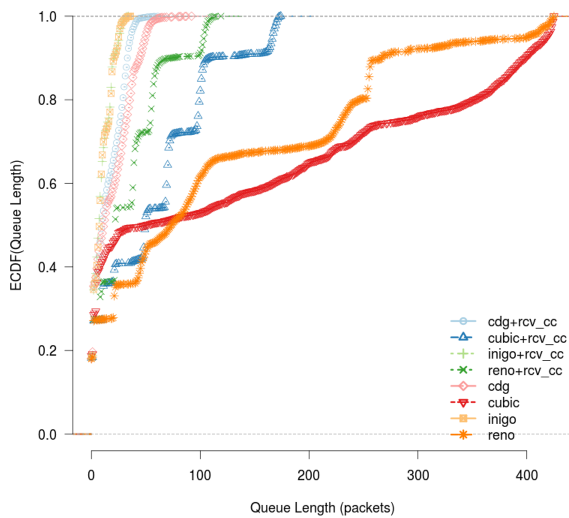


Figure 3: Five Flows Converge - Receiver Congestion Control Empirical CDF of Bottleneck Queue
Inigo sender keeps the shortest queues when ECN is unavailable, and Inigo receiver-side congestion control helps CUBIC and Reno reduce their queues by almost 4$\times$

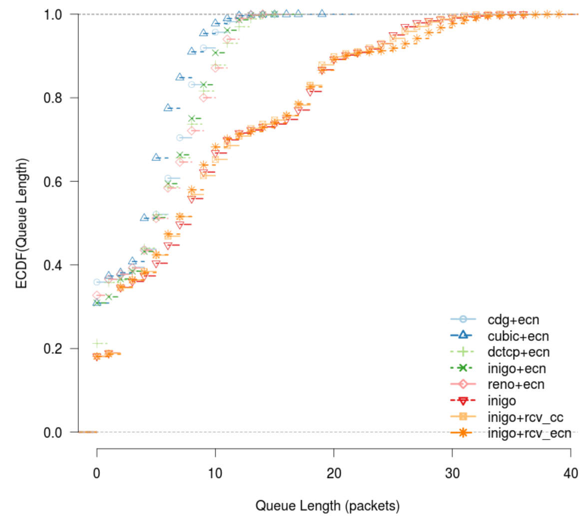


Figure 5: Five Flows Converge without ECN Empirical CDF of Bottleneck Queue
Inigo approaches the low queue length of ECN-enabled TCPs—only exceeding 20 packets approximately 10% of the time

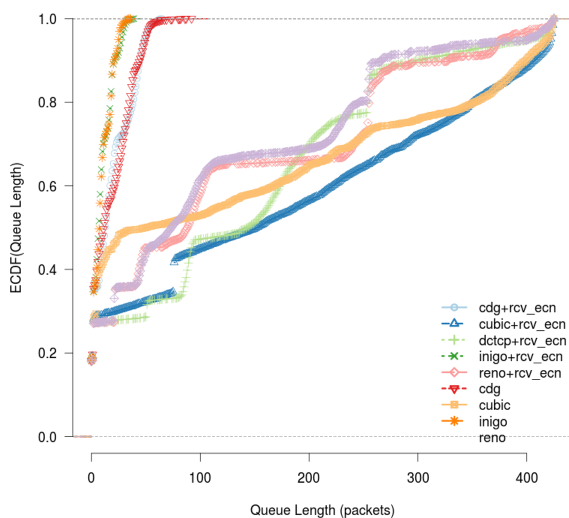


Figure 4: Five Flows Converge with Receiver ECN Empirical CDF of Bottleneck Queue
Inigo receiver-side marking appears to be non-functional

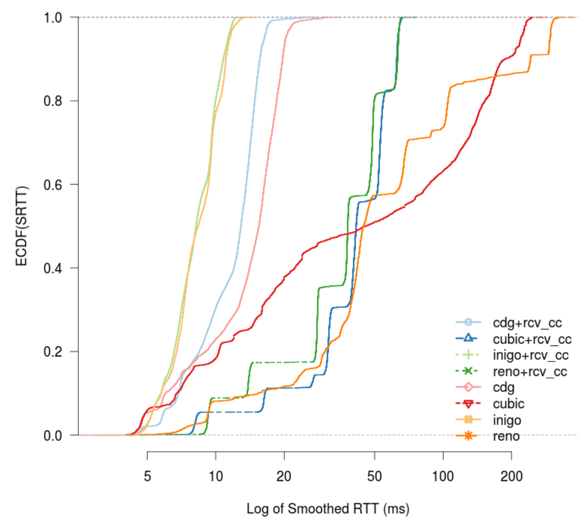


Figure 6: Five Flows Converge - Receiver Congestion Control Empirical CDF of TCP Smoothed RTT
Inigo sender's 1.0 quantile is less than half CDG's and 5$\times$ less than other TCP's 0.5 quantile while Inigo's receiver tightens up CUBIC and Reno's upper 0.5 quantile by 150 milliseconds

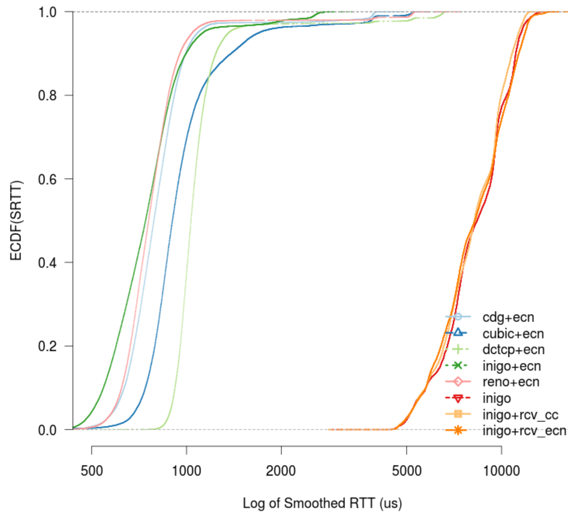
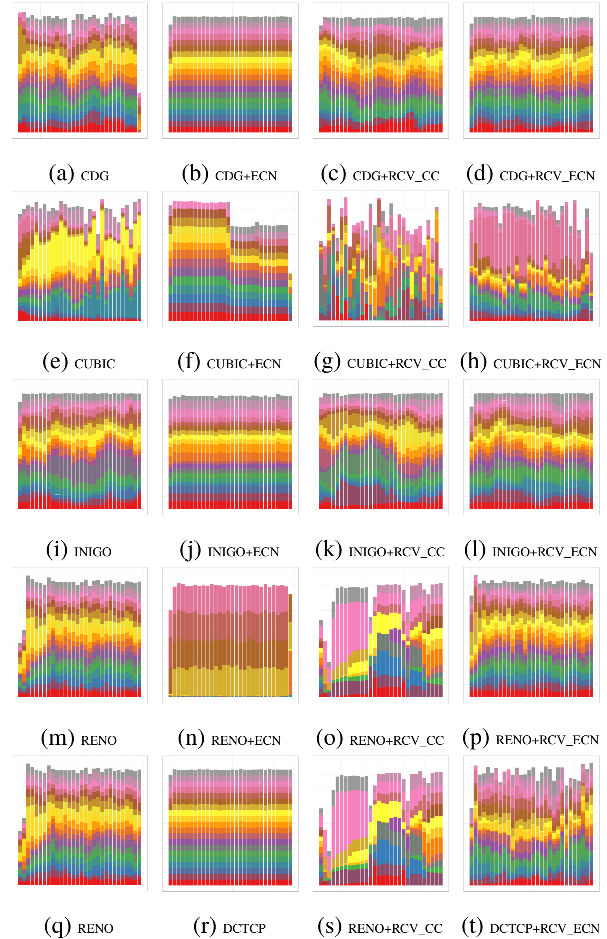


Figure 7: Five Flows Converge with ECN
Empirical CDF of TCP Smoothed RTT
SRTTs below 4 milliseconds are impossible in this scenario, so all of the ECN results should be shifted close to Inigo

switch's bottleneck queue length and TCP's SRTT. Note that the X axis of the SRTT graphs have a logarithmic scale since RTTS can exhibit a huge range (due to accumulating the effects of multiple queues). Inigo's SRTTs are almost entirely less than 10 milliseconds, whereas CDG's 1.0 quantile is around twice that and other TCPs can often incur SRTTs of more than 5× Inigo, without ECN.

Figure 7 gives the impression that Inigo's RTT-based congestion control is far outclassed by ECN-enabled TCPs. However, the minimum RTT for this scenario is four milliseconds, so the SRTTs below 4 millisecond should not be possible. Perhaps the TCP SRTT calculations are oddly effected by ECN. Regardless, the tails of ECN-enabled TCPs in the graph are more indicative of the actual RTTs being achieved. Future experiments will be constructed to verify that.

With regards to the dangers of shrinking the receiver window, we note that during this test the first CUBIC flow retransmitted 4457 segments and incurred three timeouts in the loss state. The last CUBIC flow retransmitted 989 segments and had one timeout. In contrast, when Inigo's receiver was managing the window the first CUBIC flow retransmitted 4 segments, the last flow retransmitted zero, and neither experienced a timeout. So, despite the what the RFCs recommend, it appears that shrinking the window in this manner may be preferrable to not shrinking it.



(a) CDG (b) CDG+ECN (c) CDG+RCV_CC (d) CDG+RCV_ECN
(e) CUBIC (f) CUBIC+ECN (g) CUBIC+RCV_CC (h) CUBIC+RCV_ECN
(i) INIGO (j) INIGO+ECN (k) INIGO+RCV_CC (l) INIGO+RCV_ECN
(m) RENO (n) RENO+ECN (o) RENO+RCV_CC (p) RENO+RCV_ECN
(q) RENO (r) DCTCP (s) RENO+RCV_CC (t) DCTCP+RCV_ECN

Figure 8: 20 Flow Incast - Bandwidth
Inigo 8i achieves full utilization even without ECN, where other TCPs without ECN overfill the buffer and lose packets

4.2 Incast

In our incast experiment 20 iperf2 clients start sending to one server simultaneously for 30 seconds. This scenario is not uncommon in storage systems or whenever data must be aggregated.

The two most common problems seen with incast in Figure 8 are large drops in utilization due to overflowing buffers and flows suffering from total bandwidth starvation, even in situations that appear to be consistently sharing fairly. ECN-enabled TCPs in column 8b do appear to be stabilized, but ECN is obviously not a guarantee since both CUBIC 8f and Reno 8n exhibit major issues.

Inigo+ECN 8j is more stable than other runs in row 8i, but it did lose one flow. CDG 8a appears almost as good as Inigo, but sometimes suffers from packet loss, as seen in Figure 9. In fact, only the Inigo sender is able to avoid overflowing the buffer without the benefit of

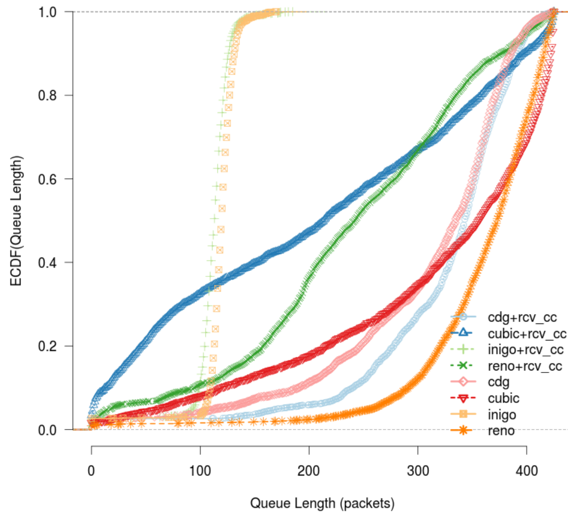


Figure 9: 20 Flow Incast - Receiver Congestion Control
Empirical CDF of Bottleneck Queue
Inigo's bottleneck queue length is capped at around 150 packets, while other TCP's without ECN experience unbounded queue growth

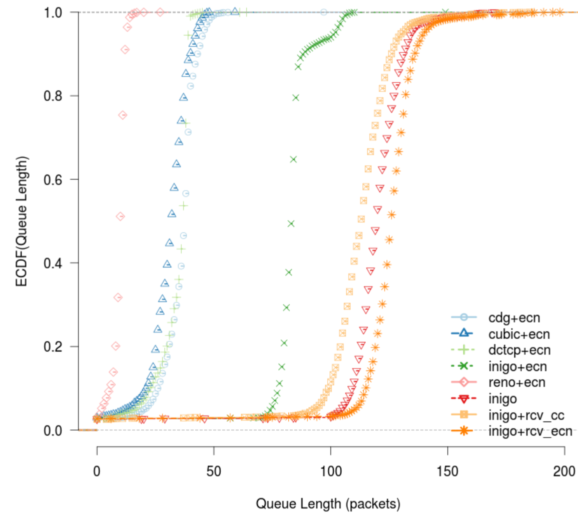


Figure 10: 20 Flow Incast without ECN
Empirical CDF of Bottleneck Queue
Inigo's queue is about 4× greater then well behaved DCTCP and CDG+ECN, but the shape of Inigo's CDF indicates that it might be able to close the gap with some tuning

ECN. Inigo's receiver-side congestion control again helps CUBIC and Reno slow their buffer growth, but fails to make a substantive difference.

Figure 9 shows Inigo's bottleneck queue length is capped at around 150 packets, while other TCP's without ECN experience unbounded queue growth. The vertical CDF of Inigo's queue length in figure 10 indicates that it might be possible to get closer to ECN-enabled behavior if Inigo is tuned to exit Slow Start slightly earlier or its maximum backoff was slightly increased to $5W/8$ instead of $W/2$. In terms of delay, Inigo's SRTT stays around 10 milliseconds, which is more than double RTT_{min}, but not out of control. Like the previous experiment, figure 10 shows the SRTTs of ECN-enabled TCPs are being miscalculated, and should be shifted three to four milliseconds to the right.

4.3 RTT Dilation

In order to test the effectiveness of RTT dilation, we use the FLExible Network Tester (Flent) [27]. This particular test, called Realtime Response Under Load, uses four different TCP variants to create simultaneous upload and downloads between a single client and server while monitoring latencies with multiple types of pings. The upload start times were staggered in order to show how TCP Inigo keeps latencies low when possible, but competes for bandwidth when low latencies are not possible.

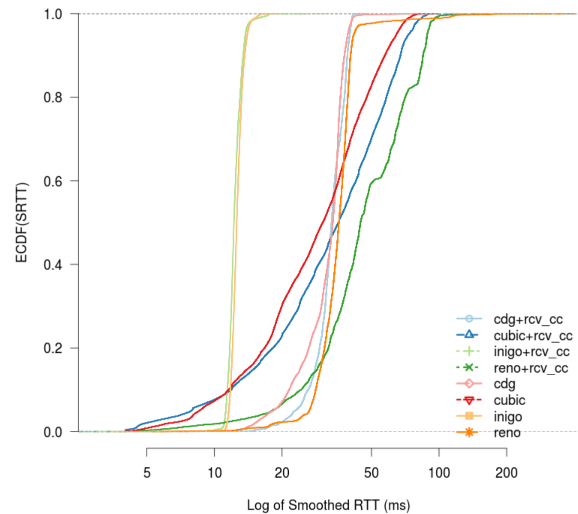


Figure 11: 20 Flow Incast - Receiver Congestion Control
Empirical CDF of TCP Smoothed RTT
Inigo's SRTT stays around 10 milliseconds—90% of all SRTTs of the other non-ECN TCPs tested exceed that by a large margin

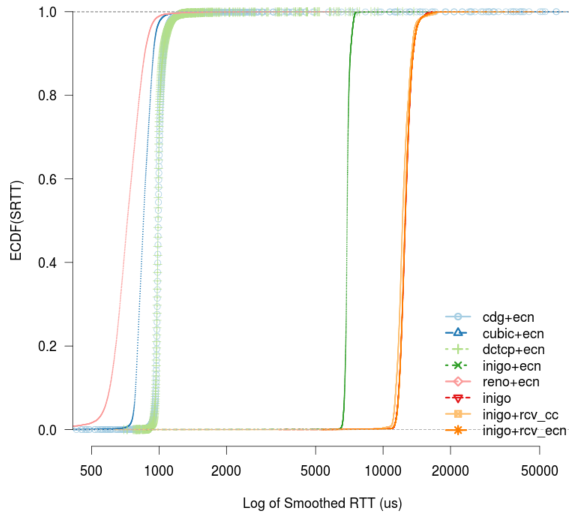


Figure 12: 20 Flow Incast with ECN
Empirical CDF of TCP Smoothed RTT
The SRTTs of ECN-enabled TCPs are being miscalculated, and should be shifted three to four milliseconds to the right

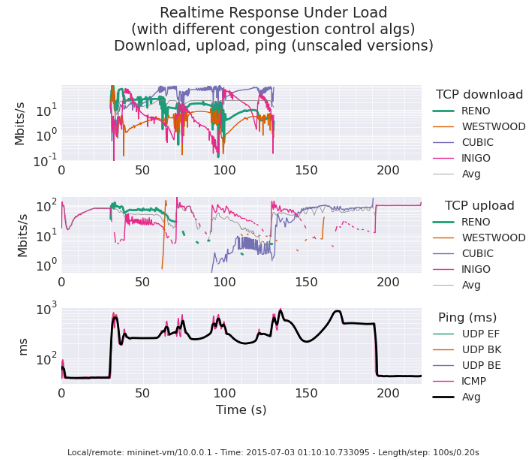


Figure 13: Inigo Realtime Response Under Load
When Inigo detects persistent congestion over multiple RTTs, it dilates RTT_{min} and fights for its share of the bandwidth. Once the congestion abates, Inigo returns to its regular buffer-friendly behavior

These were run within a Mininet environment consisting of 100Mbps links and 10ms RTT. CUBIC generally dominates all other TCPs in acquiring bandwidth.

5 Related Work

Many other enhancements have been proposed to improve or leverage DCTCP, including RTT-fairness through subwindow adjustments [2], ultra-low latency with phantom queues [3], deadline-awareness [49], minimizing flow completion times [38], sender-side only DCTCP [30], application to wireless networks [50] stability enhancements [10], elimination of Slow Start in conjunction with Data Center Bridging [46], and various deployability enhancements [44].

And academia is not alone in trying to take DCTCP further. The IETF is discussing DCTCP's vulnerability to ACK-loss, along with the ways they might improve congestion notification and DCTCP [9, 31, 16]. The enabling of ECN on all Apple systems [33] could encourage more ECN marking in routers and help make DCTCP feasible on the Internet [32]. However, those routers would need to be configured both to mark ECN as DCTCP requires and to enable DCTCP to coexist with other TCP variants. Change of that magnitude should not be expected.

DCTCP has not eliminated the interest in other congestion control algorithms in the data center or for the Internet. CAIA Delay-Gradient (CDG) TCP [26] uses minimum and maximum RTTs to reason about congestion, with an emphasis on coexistence with loss-based congestion control in wide area networks, and it was recently merged into the Linux 4.2 kernel. Like Inigo, it does not require special hardware or effort to deploy.

Remy [51, 45] has been used to generate congestion control protocols, and it compares favorably to many previous loss-based and delay-based TCPs in simulations. However, RemyCC results in RTTs $4-6\times$ worse than DCTCP since Remy does not yet take advantage of ECN or AQM. The Tao protocols in later Remy experiments appear to approach the performance of an omniscient schedule, but DCTCP was not included in that comparison. It remains to be seen if machine generated congest control is practical or if it can lead to new and better understanding of congestion.

Dong, et al. make the argument that even though Remy generates protocols, it searches a space of hardwired responses to packet level events, and its performance can degrade when the real network does not match its assumptions, just like most TCPs [18]. They propose Performance-oriented Congestion Control (PCC), a sender-side modification to TCP that controls its rate based on continuous experimental trials of rates differing $1-5\%$. PCC makes fewer assumptions than most congestion control algorithms, but one it shares is that repeatedly trying higher rates is necessary even if they always lower the measured utility.

Inigo, like DCTCP, does not respond to packet level events, but a congestion ratio collected over a set of packets. In fact, the congestion ratio is a measure of utility

that means “keep the link full, and latency low”. The assumption that increases in RTTs are due to congestion is generally valid, with the primary exception being operating system noise. Inigo’s use of a congestion response is inherently robust to noise, see § 2.5. Rare situations such as being forced mid-flow to use a longer route or the underlying media enabling Forward Error Correction may cause Inigo to need to detect a new minimum RTT, and RTT dilation handles corner cases such as those. While a PCC prototype has been made publicly available, we have not yet compared it to Inigo.

Lee, et al. propose DX [34], which shows that accurate queue delay measurements can be attained even for high speed networks by modifying drivers and adding TCP options. Their congestion response is driven by the ratio of the measured average queuing delay to an estimate of the number of competing flows, resulting in higher utilization and lower latency than DCTCP. TIMELY [36] uses hardware timestamps, delay gradients, and rate control to implement congestion control for RDMA traffic. While this paper does not include a direct comparison with DX or TIMELY, it is reasonable to expect that the RTT-based congestion control proposed here would also benefit from improved timestamps. However, the congestion control described in this paper can be used without any additional development effort and on any hardware.

Change may be well worth it in some cases, but networks tend to resist change. Consider the slow uptake of IPV6, ECN, RED [21], and FQ_Codel [39]. Even when hardware and software support became common, configurations did not change quickly (or at all) to take advantage of them. Another example is Quantized Congestion Notification (QCN) [20], which became a standard along with other Data Center Bridging technologies. The only QCN-enabled hardware the authors of this paper are aware of comes from Mellanox, and that is likely because they support similar features in their Infiniband products. Norm Finn, the editor of the QCN standard, gave the following statement (personal communication, January 16, 2014) when asked why it was hard to find QCN-enabled hardware:

> “Judging by the scarcity of implementations of IEEE Std 802.1Qau, the principle benefit obtained from the standard may not have been congestion control, itself. The promise of congestion control made more palatable the standardization of IEEE Std 802.1Qbb Priority Flow Control, to which objections were raised on the grounds that it could cause a deadlock. 802.1Qau lessens the likelihood of 802.1Qbb deadlocks.”

6 Conclusion

The difficulty inherent in deploying new technology on networks provided part of the motivation for the TCP congestion control variant, Inigo, described in this paper. Inigo does not require special hardware, driver development, or switch configurations. But if enhanced timestamping does become generally available, then Inigo will automatically benefit.

Inigo’s sender-side RTT-based congestion control integrates with DCTCP and provides a fallback that resembles DCTCP’s ECN-based behavior. The receiver-side RFD-based congestion control, though less effective than the sender-side due to coarse-grained timestamps, is able to encourage fair bandwidth sharing and smaller buffer occupancy of TCP senders such as CUBIC and Reno. We refer to both of these modifications as TCP Inigo in this paper, even though each modification can be brought into service separately.

Inigo is still in an early state and will require much more testing before it can be confidently deployed. Among other things Inigo’s ECN marking should be fixed, since it could help make use of ECN ubiquitous on the Internet. Then switches will not need to be configured to do the marking themselves. The receiver side congestion control needs to be tested against more TCP variants, especially against those of other OSes.

Acknowledgments

Thanks to Los Alamos National Laboratory and to Google for partially funding this work with a Research Award.

References

[1] ALIZADEH, M., GREENBERG, A., MALTZ, D. A., PADHYE, J., PATEL, P., PRABHAKAR, B., SENGUPTA, S., AND SRIDHARAN, M. Data center tcp (dctcp). *ACM SIGCOMM computer communication review 41*, 4 (2011), 63–74.

[2] ALIZADEH, M., JAVANMARD, A., AND PRABHAKAR, B. Analysis of dctcp: stability, convergence, and fairness. In *Proceedings of the ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems* (2011), ACM, pp. 73–84.

[3] ALIZADEH, M., KABBANI, A., EDSALL, T., PRABHAKAR, B., VAHDAT, A., AND YASUDA, M. Less is more: trading a little bandwidth for ultra-low latency in the data center. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation* (2012), USENIX Association, pp. 19–19.

[4] BLUMENTHAL, M. S., AND CLARK, D. D. Rethinking the design of the internet: the end-to-end arguments vs. the brave new world. *ACM Transactions on Internet Technology (TOIT) 1*, 1 (2001), 70–109.

[5] BORMAN, D., SCHEFFENEGGER, R., AND JACOBSON, V. Rfc 7323: Tcp extensions for high performance. https://tools.ietf.org/html/rfc7323, 2014.

[6] BRADEN, R. Rfc 1122: Requirements for internet hosts. https://tools.ietf.org/html/rfc1122, 1989.

[7] Briscoe, B., Brunstrom, A., Petlund, A., Hayes, D., Ros, D., Tsang, J., Gjessing, S., Fairhurst, G., Griwodz, C., and Welzl, M. Reducing internet latency: a survey of techniques and their merits. *IEEE Communications Surveys & Tutorials* (2014).

[8] Briscoe, B., and De Schepper, K. Scaling tcp's congestion window for small round trip times. Tech. rep., Technical report TR-TUB8-2015-002, BT, 2015.

[9] Briscoe, B., and Mathis, M. Congestion exposure (conex) concepts, abstract mechanism and requirements. https://tools.ietf.org/html/draft-ietf-conex-abstract-mech-13, 2014.

[10] Chen, W., Cheng, P., Ren, F., Shu, R., and Lin, C. Ease the queue oscillation: Analysis and enhancement of dctcp. In *Distributed Computing Systems (ICDCS), 2013 IEEE 33rd International Conference on* (2013), IEEE, pp. 450–459.

[11] Corbet, J. Jls2009: Generic receive offload. https://lwn.net/Articles/358910/.

[12] Corbet, J. Network transmit queue limits. https://lwn.net/Articles/454390/.

[13] Corbet, J. Tcp segmentation offloading. https://lwn.net/Articles/9123/.

[14] Corbet, J. Tcp small queues. https://lwn.net/Articles/507065/.

[15] Corbet, J. Tso sizing and the fq scheduler. http://lwn.net/Articles/564978/.

[16] De Schepper, K., Bondarenko, O., Tsang, J., and Briscoe, B. Data centre to the home: Ultra-low latency for all (under submission). http://www.bobbriscoe.net/projects/latency/dctth_preprint.pdf, 2015.

[17] Dean, J., and Barroso, L. A. The tail at scale. *Communications of the ACM 56*, 2 (2013), 74–80.

[18] Dong, M., Li, Q., Zarchy, D., Godfrey, P. B., and Schapira, M. Pcc: Re-architecting congestion control for consistent high performance. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)* (Oakland, CA, May 2015), USENIX Association, pp. 395–408.

[19] Dumazet, E. tcp: switch rtt estimations to usec resolution. https://goo.gl/TtBZ3Z, 2014.

[20] Finn, N. Ieee standard for local and metropolitan area networks—virtual bridged local area networks - amendment: 10: Congestion notification. http://www.ieee802.org/1/pages/802.1au.html, 2010.

[21] Floyd, S., and Jacobson, V. Random early detection gateways for congestion avoidance. *Networking, IEEE/ACM Transactions on 1*, 4 (1993), 397–413.

[22] Gettys, J., and Nichols, K. Bufferbloat: Dark buffers in the internet. *Queue 9*, 11 (2011), 40.

[23] Ha, S., and Rhee, I. Taming the elephants: New tcp slow start. *Computer Networks 55*, 9 (2011), 2092–2110.

[24] Ha, S., Rhee, I., and Xu, L. Cubic: a new tcp-friendly high-speed tcp variant. *ACM SIGOPS Operating Systems Review 42*, 5 (2008), 64–74.

[25] Handigol, N., Heller, B., Jeyakumar, V., Lantz, B., and McKeown, N. Reproducible network experiments using container-based emulation. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies* (2012), ACM, pp. 253–264.

[26] Hayes, D. A., and Armitage, G. Revisiting tcp congestion control using delay gradients. In *NETWORKING 2011*. Springer, 2011, pp. 328–341.

[27] Høiland-Jørgense, T. Flent: The flexible network tester. https://flent.org/.

[28] Jacobson, V., Braden, R., and Borman, D. Rfc 1323: Tcp extensions for high performance. https://tools.ietf.org/html/rfc1323, 1992.

[29] Jain, M., and Dovrolis, C. Pathload: A measurement tool for end-to-end available bandwidth. In *In Proceedings of Passive and Active Measurements (PAM) Workshop* (2002), Citeseer.

[30] Kato, M. Improving transmission performance with one-sided datacenter tcp. Master's thesis, Keio University, 2014.

[31] Kühlewind, M., Scheffenegger, R., and Briscoe, B. Rfc 7560: Problem statement and requirements for increased accuracy in explicit congestion notification (ecn) feedback. https://tools.ietf.org/html/rfc7560, 2015.

[32] Kühlewind, M., Wagner, D. P., Espinosa, J. M. R., and Briscoe, B. Using data center tcp (dctcp) in the internet. In *Globecom Workshops (GC Wkshps), 2014* (2014), IEEE, pp. 583–588.

[33] Lakhera, P. Your app and next generation networks. http://goo.gl/UEHYtq, 2015. Apple.

[34] Lee, C., Park, C., Jang, K., Moon, S., and Han, D. Accurate latency-based congestion feedback for datacenters. In *USENIX ATC* (2015), vol. 15.

[35] Miller, D., Hemminger, S., et al. [patch] make cubic hystart more robust to rtt variations. http://thread.gmane.org/gmane.linux.network/188738/focus=188808, 2011.

[36] Mittal, R., Dukkipati, N., Blem, E., Wassel, H., Ghobadi, M., Vahdat, A., Wang, Y., Wetherall, D., Zats, D., et al. Timely: Rtt-based congestion control for the datacenter. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (2015), ACM, pp. 537–550.

[37] Morton, J., and Taht, D. Cake - common applications kept enhanced. http://www.bufferbloat.net/projects/codel/wiki/CakeTechnical, 2016. http://www.bufferbloat.net/projects/codel/wiki/Cake.

[38] Munir, A., Qazi, I. A., Uzmi, Z. A., Mushtaq, A., Ismail, S. N., Iqbal, M. S., and Khan, B. Minimizing flow completion times in data centers. In *INFOCOM, 2013 Proceedings IEEE* (2013), IEEE, pp. 2157–2165.

[39] Nichols, K., and Jacobson, V. Controlling queue delay. *Communications of the ACM 55*, 7 (2012), 42–50.

[40] Parsa, C., and Garcia-Luna-Aceves, J. J. Improving TCP congestion control over internets with heterogeneous transmission media. In *Proceedings of the 7th IEEE International Conference on Network Protocols (ICNP)* (1999), IEEE.

[41] Postel, J. Rfc 793: Transmission control protocol. usc. *Information Sciences Institute 27*, 793 (1981), 123–150.

[42] Ramakrishnan, K., Floyd, S., Black, D., et al. Rfc 3168: The addition of explicit congestion notification (ecn) to ip. *Network Working Group, IETF*, 3168 (2001).

[43] Saltzer, J. H., Reed, D. P., and Clark, D. D. End-to-end arguments in system design. *ACM Transactions on Computer Systems (TOCS) 2*, 4 (1984), 277–288.

[44] Shewmaker, A., Maltzahn, C., Obraczka, K., and Brandt, S. Tcp inigo: Fighting congestion with both hands. Tech. rep., UC Santa Cruz, 2014.

[45] Sivaraman, A., Winstein, K., Thaker, P., and Balakrishnan, H. An experimental study of the learnability of congestion control. In *Proceedings of the 2014 ACM conference on SIGCOMM* (2014), ACM, pp. 479–490.

[46] Stephens, B., Cox, A. L., Singla, A., Carter, J., Dixon, C., and Felter, W. Practical dcb for improved data center networks. In *INFOCOM, 2014 Proceedings IEEE* (2014), IEEE, pp. 1824–1832.

[47] Taht, D. Implementing comprehensive queue management on home routers. IETF, 2014.

[48] Tirumala, A., Qin, F., Dugan, J., Ferguson, J., and Gibbs, K. Iperf: The tcp/udp bandwidth measurement tool. http://dast.nlanr.net/Projects, 2005.

[49] Vamanan, B., Hasan, J., and Vijaykumar, T. Deadline-aware datacenter tcp (d2tcp). *ACM SIGCOMM Computer Communication Review 42*, 4 (2012), 115–126.

[50] Wang, J., Jiang, Y., Ouyang, Y., Li, C., Xiong, Z., and Cai, J. Tcp congestion control for wireless datacenters. *IEICE Electronics Express 10*, 12 (2013), 20130349–20130349.

[51] Winstein, K., and Balakrishnan, H. Tcp ex machina: Computer-generated congestion control. In *ACM SIGCOMM Computer Communication Review* (2013), vol. 43, ACM, pp. 123–134.

A Availability

The kernel module implementing the RTT-based fallback for DCTCP, as well as the receiver-side congest control patch, together called TCP Inigo in this paper, the experimental results in this paper, and the Mininet experiment framework can be downloaded from GitHub.

`https://github.com/systemslab/tcp_inigo`

Inigo should be easily back ported to earlier kernels, although the effectiveness of the sender side will be strongly affected by the decreased RTT resolution before Linux 3.14, among other changes. The receiver-side modification is mostly contained in two functions, inserted before ECN processing and a seven line change to the receive window size selection code. Although the location of the new function calls will be slightly different prior to DCTCP's inclusion in Linux 3.17, the impact to Inigo with prior kernels should only effect DCTP senders. Of course, DCTCP's receiver-side change could be backported too.

B TSO

The Linux `pkts_acked` `tcp_congestion_ops` hook is passed a socket pointer, the number of packets being ACKed, and an RTT in microseconds. If TSO [13] is enabled, then the TCP stack will send multiple segments worth of data at a time to the lower layers of the network stack, therefore leaving TCP with only one RTT measurement for multiple packets.

Given a RTT that exceeds $RTT_{min} + d_{thresh}$ (i.e. a late RTT), one could try to extract more information than simply incrementing the $RTT_{observations}$ and RTT_{late} each by one as in algorithm 2. In the worst case one could assume all of the segments are sent onto the network as a single burst, and therefore all are late. A slightly less pessimistic assumption would be that each RTT was barely late, so the total delay minus *dthresh* would give the number of late RTTS. Alternatively if packet pacing is assumed, then the measured delay would be equally shared amongst all of the packets being ACKed.

Algorithm 10 uses both of those lines of reasoning.

Algorithm 10 RTT Congestion Marking for TSO and Packet Pacing

for each ACK **do**
  $RTTs_{observed} \leftarrow RTTs_{observed} + pkts_acked$
  if $RTT \geq RTT_{min} + d_{thresh}$ **then**
    $delay \leftarrow RTT - RTT_{min}$
    if $pacing \wedge (delay/pkts_acked > d_{thresh})$ **then**
      $RTTs_{late} \leftarrow RTTs_{late} + pkts_acked$
    else
      $pkts_late \leftarrow min(pkts_acked, delay/d_{thresh})$
      $RTTs_{late} \leftarrow RTTs_{late} + pkts_late$
    end if
  end if
end for

C RFD Discussion

The simulator implementation of TCP Santa Cruz required modifications to both the sender and receiver, and results showed promise, but it was never tested on real networks. This was evidently due in part to TCP Santa Cruz's reliance on an experimental TCP option, unlike this work.

Others have also used RFD to reason about bandwidth and congestion. Pathload [29] used packet trains to probe the available bandwidth of a network. HyStart [23] found Pathload's techniques unsuitable for integration with TCP, but used them as inspiration for its ACK-train heuristic used as a signal to exit Slow Start.

The receiving side of TCP can use timestamps to calculate RFD, but unfortunately the existing TCP timestamps are too coarse-grained for data centers. RFC 1323 and the updated RFC 7323 [28, 5] both recommend a timestamp resolution between 1 millisecond and 1 second per tick, whereas data center RTTs are measured in microseconds. Similarly unfortunate, the receiver only has an estimate of the RTT in milliseconds, and it appears to be less than the actual RTT in our experiments. This will tend to magnify the measurement of congestion since the minimum RTT is used to define d_{thresh}.

In order to accomodate both Intenet and data center latencies, TCP could keep track of minimum $S_{i,j}$ and $R_{i,j}$ for consecutive packets. If RTTs and timestamp deltas

for both sender and receiver are less than or equal to one millisecond, then TCP could swap out the millisecond timestamp operations for microsecond versions. Relying on both sides being able to increase timestamp resolution would be the sort of change that would inhibit adoption. Also, a side effect of increasing the timestamp resolution would be to reduce opportunities for Generic Receive Offload [11].