TCP Inigo:
Ambidextrous Congestion Control

Andrew G. Shewmaker [*], Carlos Maltzahn [**], Katia Obraczka [**], Scott Brandt [**], and John Bent [***]

[*]Los Alamos National Laboratory and UC Santa Cruz
[**]UC Santa Cruz
[***]Seagate Government Solutions

Abstract

No one likes waiting in traffic, whether on a road or on a computer network. Stuttering audio, slow interactive feedback, and untimely pauses in video annoy everyone and cost businesses sales and productivity. An ideal network should (1) minimize latency, (2) maximize bandwidth, (3) share resources according to a desired policy, (4) enable incremental deployment, and (5) minimize administrative overhead. Many technologies have been developed, but none yet satisfactorily address all five goals. The best performing solutions developed so far require controlled environments where coordinated modification of multiple components in the network is possible, but they suffer poor performance in more complex scenarios.

We present TCP Inigo, which uses independent delay-based algorithms on the sender and receiver (i.e. ambidextrously) to satisfy all five goals. In networks with single administrative domains, like those in data centers, Inigo's fairness, bandwidth, and latency indices are up to 1.3× better than the best deployable solution. When deployed in a more complex environment, such as across administrative domains, Inigo possesses latency distribution tail up to 42× better.

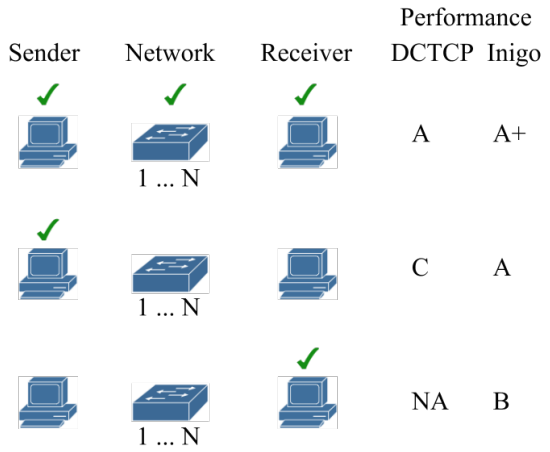

Figure 1: **Mininet experiments show Inigo's latencies are up to 1.3× better than DCTCP, the best deployable solution, when all components of a network are properly configured (green check). Inigo's sender-only mode has a latency index is up 42× better than DCTCP's corresponding failure mode; and the Inigo receiver can also improve the performance of other TCP senders.** Letter grades are relative to a C for Reno-level performance.

1 Introduction

Congested networks remain a perennial concern in data centers and the Internet. For businesses, the long tail of variations in delay can cost money [15], and congestion made worse by bufferbloat [19] creates pain for every-day users. Network congestion can even cause performance collapse in a worst case scenario such as incast in a storage network. Additionally, the network will come under more pressure as the number of users and the speed of storage increase.

In Section § 2 we overview specific techniques that improve performance in controlled environments, such as data centers, where all components (i.e. end-hosts and middle-boxes) can be modified. But the effectiveness of those same techniques significantly degrades in uncontrolled environments when interacting with unmodified components across network borders. Established systems with multiple owners and long histories, like most networks, favor incremental evolution over dramatic change because upgrades are costly and decisions to upgrade are made independently. Even data centers and supercomputers, which are prime examples of scenarios where new network technologies can be leveraged, must communicate frequently with external systems, and end-to-end arguments [5, 39] should be considered.

There is considerable practical value in being able to

improve network performance while minimizing the effort needed to deploy and maintain the changes. Figure 1 shows just a few examples of the many failure modes a solution to congestion should handle.

Inigo includes two primary contributions to the state-of-the art. First, a sender-only modification inspired by DCTCP [1, 4] but uses Round-trip-times (RTTs) to mimic Explicit Congestion Notification (ECN). Second, a receiver-only modification that similarly mimics ECN with differences in One Way Delay (OWD). Inigo's worst-case performance better is than DCTCP's because Inigo's sender and receiver modifications are delay-based and can operate independently or together. The first row in Figure 1 represents a best-case administrative scenario, such as a data center, in which every component in the network is under a single authority and can be upgraded and configured coherently. The second and third rows illustrate worst-case scenarios in which every part of the network is owned by a different entity only one end-host can be modified.

This paper focuses on ways to improve network performance for applications built on TCP. While other protocols exist, TCP is the most widely used. Furthermore, the techniques Inigo uses can be re-applied to other protocols that track RTTs or use timestamps at end-hosts.

The rest of this paper is organized as follows: First, more background in congestion control and related work is described (§ 2). Next, we describe the Inigo sender-side (§ 3) and receiver-side (§ 4). We then evaluate Inigo (§ 5) and demonstrate the effectiveness of both techniques independently and combined. Finally, we conclude (§ 6) and describe the availability of TCP Inigo code and Mininet experiments (§ 7).

2 Background and Related Work

The Transmission Control Protocol (TCP) is the broadest deployed *transport protocol* ensuring reliable delivery of data. TCP endures despite its suboptimal performance in various scenarios because it assumes little about the details of the networks it traverses yet is able to provide adequate performance in most cases, as long as packet loss generally corresponds with congestion instead of the reliability of the connection.

Standard ECN support [38] directs a sender to halve its window once per RTT upon seeing an acknowledgment (ACK) marked with Congestion Exists (CE), whereas DCTCP [1] tracks the ratio of bytes marked with CE to the total number of bytes ACKed in order to estimate the extent of congestion. A congestion ratio of 1 causes DCTCP to halve its window, and smaller ratios cause it to back off correspondingly 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 [28], but it compromises the performance of DCTCP when the receiver has been modified. New TCP header options could allow a sender to detect if a receiver has been modified, but that type of solution can run into problems when middle-boxes manipulate headers without properly supporting new or rarely used options. The primary tool at present for ensuring DCTCP senders talk to modified receivers is to configure per-route congestion control. That works for homogeneous subnets, but it is an increasingly complex and infeasible solution when communication occurs between a wide variety of hosts controlled by other organizations.

Switches must also be configured to mark ECN appropriately for use with DCTCP. Configuring for DCTCP is simpler than for RED [18], and it can use the common support for RED 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, they may consider configuring separate switch queues to be impractical, or setting per-route congestion control on the application side may not be fine-grained enough for the applications running on their cloud.

Change may be well worth it in some cases, but networks tend to resist change. Consider the slow uptake of IPV6, ECN, RED, and FQ_CoDel [35]. Even when hardware and software support become common, network administrators and application developers do not change their configurations quickly (or at all) to take advantage of them.

Many enhancements have been proposed to improve or leverage DCTCP, including RTT-fairness through sub-window adjustments [2], ultra-low latency with phantom queues [3], deadline-awareness [44], minimizing flow completion times [34], sender-side only DCTCP [28], application to wireless networks [45] stability enhancements [10], elimination of Slow Start in conjunction with Data Center Bridging [42], and various ideas for deployability enhancements [40].

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, 29, 14]. The enabling of ECN on all Apple systems [31] could encourage more ECN marking in routers and help make DCTCP feasible on the Internet [30]. 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 [23] 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 merged into the Linux 4.2 kernel. It is a sender-side only modification, so is easy to deploy.

Remy [46, 41] 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 hard-wired responses to packet level events, and its performance can degrade when the real network does not match its assumptions, just like most TCPs [16]. They propose Performance-oriented Congestion Control (PCC), a sender-side modification to TCP that modifies its rate and packet pacing based on continuous experimental trials of rates differing 1-5%.

Lee, et al. propose DX [32], which shows that accurate queue delay measurements can be attained even for high speed networks by modifying drivers, adding TCP options, and modifying both senders and receivers. 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. The combination of these changes is almost impossible to implement in reality since broad support for new TCP header options is difficult to attain and diverse networks cannot be expected to have compliant hosts.

3 TCP Inigo Sender

TCP Inigo is composed of two independent techniques. The first is a sender-side only modification that uses TCP RTT measurements. The second uses differences in One Way Delays (OWDs) on the receiver-side. 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.

The Inigo sender-side congestion control module uses the Linux kernel's pluggable congest control interface, and has been made possible by several developments in the Linux kernel since the original DCTCP paper [1]. Internal buffer bloat and delay variability were much improved with features such as Byte Queue Limits [11], TCP Small Queues [12], and TCP Segmentation Offload (TSO) sizing and pacing [13]. Importantly, the units of the sender's RTT measurement changed from milliseconds to microseconds in 2014 [17]. These changes improve the behavior of every TCP, but they are vital for Inigo's delay-based algorithms.

RTTs are readily available measurements, but TCP's timestamps 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. If the goal is to minimize the end-to-end delay variability observed by the application layer, then the fact that the TCP RTT includes delays due to Operating System (OS) buffers and network buffers may be an advantage. Also, RTT measurements do 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. Since location and presence of congestion is combined in a marking, the end hosts cannot tell for certain how bad the congestion is.

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.

3.1 RTT Congestion Ratio

DCTCP's congestion ratio, $\alpha_{ECN} = \frac{bytes_{ECN}}{bytes_{total}}$, is driven by an ECN marking threshold designed to balance conflicting requirements—to fully utilize bandwidth while keeping latency low. In this section we will explain how we leverage the same reasoning that led to DCTCP's ECN marking threshold to justify the RTT threshold that drives Inigo's congestion ratio, $\alpha_{RTT} = \frac{RTTs_{late}}{RTTs_{observed}}$.

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

Alizadeh, et al. [2] derived equation (1), in which C and d denote the bottleneck capacity (packets/second) and RTT delay (seconds), 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. Intuitively, this threshold says that the queue should absorb bursts of up to 17% of the bandwidth-delay product before it starts telling flows to slow down.

Inigo uses *late RTTs* in the same way that DCTCP uses ECN markings to calculate and respond to the extent of congestion. Increases in RTTs are generally due to

congestion in current systems where the OS is not CPU bound and it keeps its internal bufferbloat under control.

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

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

Since the RTT signal arrives at the same frequency as ECN markings (i.e. every ACK), and since TCP RTTs correspond to increased queuing, we assume that the recommended DCTCP threshold is valid for defining what makes an RTT late. The corresponding queuing delay threshold, d_{thresh}, is simply K divided by the bottleneck capacity C. Substituting equation (1) into (2) gives (3).

Algorithm 1 Inigo RTT Congestion Marking

```
for each ACK do
    if RTT_min = 0 ∨ RTT < RTT_min then
        RTT_min ← RTT
    end if
    RTTs_observed ← RTTs_observed + 1
    if RTT ≥ RTT_min + d_thresh then
        RTTs_late ← RTTs_late + 1
    end if
end for
```

Algorithm 2 Inigo RTT Congestion Ratio

```
for every window do
    F ← RTTs_late / RTTs_observed
    α_RTT ← (1 − g) × α_RTT + g × F
    RTTs_observed ← 0
    RTTs_late ← 0
end for
```

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 of ECN-marked bytes is tracked during a window and used to update the exponential weighted moving average of the DCTCP congestion ratio α_{ECN}.

There are some subtle implications of using a congestion ratio driven by RTT observations instead of ECN marked bytes. Most importantly, RTTs allow a sender-only modification. Also, Inigo does not need to compensate for delayed ACKs since they only reduce the number of observations driving the congestion ratio and do not distort the magnitude of the congestion signal, as they do for ECN markings. Similarly, the number of RTT measurements can be further reduced when the lower layers of the network stack or hardware implement segmentation offloading, which aggregates TCP's segments into larger chunks before sending them out onto the network.

3.2 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.

Interestingly, the DCTCP Internet-Draft [4] does not specify Slow Start behavior, and the 4.4 Linux DCTCP implementation appears to overshoot the ideal congestion window and cause unnecessarily high RTTs. For this reason, when using ECN, Inigo exits Slow Start immediately upon seeing the first ACK in a window with a CE marking. Other than that, Inigo behaves the same as DCTCP when ECN is available. Matching the efficiency of an ECN-driven Slow Start exit using only delay is challenging, with HyStart [20] probably the most successful example of the technique.

Linux kernel implementations of HyStart in TCP CUBIC [21] and CDG [23] contain 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 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 [33]. Interestingly, CUBIC's new threshold is within 1.8% of the initially recommended threshold for DCTCP [1].

Algorithm 3 Inigo Slow Start

```
for every ACK do
    if cwnd ≤ ssthresh ∧ samples ≥ 8 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
```

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 3 uses $RTTs_{observed}$ and $RTTs_{late}$ from algorithm 2. If the congestion ratio is non-zero once eight RTTs are observed, then it reduces *cwnd* by the congestion ratio similarly to algorithm 4 in § 3.3. Finally it assigns $ssthresh = cwnd$.

3.3 Congestion Avoidance and Response

Alizadeh, et al. proposed an RTT-fairness enhancement [2], in which DCTCP would respond to congestion every ACK. The improvement counter-acts 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 it is reasonable to use a quicker response when the sum of the adjustments are designed to approximate the once-per-RTT response.

As an analogy, 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. While packets do not care about sudden changes in acceleration, a person feels it in the form of long tail latencies. Fortunately, 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 short-lived flows to enter and leave with smaller perturbations to long-lived flows.

Unfortunately, DCTCP's RTT-fairness update algorithm needs to adjust the window by a fraction of a packet, and implementations of DCTCP in kernel-space require the use of whole integer variables. Alternatively, the sender's window could be tracked in bytes like 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 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 4. We observed (§ 5) 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 link utilization. It does not significantly alter the DCTCP fluid model analysis [2] since its impact on the average per-flow window size is small. At most, it causes flows to operate more often in the primary operating regime.

Algorithm 4 Inigo RTT-fairness Congestion Response

```
for every W_sub ACKs, where 1 < W_sub < W do
    if α > 0 then
        W ← W − W_sub α/2
    end if
end for
for every window do
    W ← W + 2
end for
```

Also, since two is much smaller than the largest sensible window for an unloaded path, incrementing by two satisfies the additive increase policy stated by Jacobsen [24]. In appendix D Jacobson elaborated on the choice of the 1-packet increase, saying that the goal was to limit the average loss rate caused by gently probing for bandwidth to <1%. While the 1-packet increase was on the aggressive side for Arpanet with its 4-5 packet largest sensible window, later TCPs such as CUBIC showed the need for more aggressive probing on long, fat networks. Furthermore, Inigo can afford a more assertive probe for bandwidth since it uses delay to keep bottleneck buffers low and is unlikely to cause packet loss.

4 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). The receiver controls congestion in a style similar to DCTCP via the receive window. Theoretical benefits of receiver-side congestion control include: (1) switch configuration is unnecessary, (2) TCPs senders are forced to use a maximum window size calculated by a single algorithm instead of a variety of algorithms, and (3) mistaken maximum windows sizes due to ACK loss are corrected upon next ACK.

4.1 Relative Forward Delay

Relative Forward Delay (RFD) was defined as the difference of OWDs by Parsa, et al. [36] when they used it in

the congestion control of TCP Santa Cruz. 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 [26] used packet trains to probe the available bandwidth of a network. HyStart [20] 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 [25, 6] 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}.

Algorithm 5 RFD Congestion Marking

for each packet received **do**
  if $RTT_{min} = 0 \vee RTT < RTT_{min}$ **then**
    $RTT_{min} \leftarrow RTT$
  end if
  $bytes_{total} \leftarrow bytes_{pkt}$
  $S_{i,j} \leftarrow tsval_j - tsval_i$
  $R_{i,j} \leftarrow tsecr_j - tsecr_i$
  if $S_{i,j} = 0 \wedge R_{i,j} = 0$ **then return**
  end if ▷ low clock resolution
  $D^F_{i,j} \leftarrow R_{i,j} - S_{i,j}$
  $D^F_{total} \leftarrow max(0, D^F_{total} + D^F_{i,j})$
  if $D^F_{total} \geq d_{thresh}$ **then**
    $bytes_{late} \leftarrow bytes_{pkt}$
  end if
  $tsval_i \leftarrow tsval_j$
  $tsecr_i \leftarrow tsecr_j$
end for

Algorithm 5 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 using TCP timestamp value, (*tsval*), and timestamp echo reply (*tsecr*) fields from the header. Inigo keeps earlier timestamps until the clock resolution allows differences between sends and receives to be detected. 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. The receiver's congestion ratio is updated using $\frac{bytes_{late}}{bytes_{total}}$ similarly to Algorithm 2.

The receiver tracks the congestion ratio and modifies the receive window in a fashion similar to algorithms 3 and 4, except in bytes and with an immediate ACK sent every congestion window change. Because the algorithm is so similar, it is not included in this paper. The receiver exits Slow Start immediately when D^F_{total} exceeds d_{thresh} instead of waiting for multiple measurements like the sender. This is partly because the magnitude of RFD between consecutive packets is naturally much smaller than that of RTTs, and partly because the millisecond granularity of TCP timestamps is not adequate for measuring RFD between consecutive packets sent at a high rate.

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 [37, 7]. However, 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 in Section § 5 with Inigo show that shrinking the receive window carefully results in fewer retransmitted segments than would normally occur.

Essentially, because Linux receivers are already lying to the sender about having 4× less buffer space than in reality, Inigo's small DCTCP-style adjustments, and frequent ACKs are in little to no danger of causing in-flight packets to be dropped. Other TCP stacks 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.

5 Experiments

Next we present convergence and incast experiments based on those commonly found in papers such as DCTCP [1]. We only compare Inigo to DCTCP, but our experiments verify that the goodput and latency distributions that other TCP variants can achieve is significantly worse than Inigo. In particular, we were unable to achieve acceptable results with PCC in our experiments. Also the source code to DX is unavailable, so it was not included.

Mininet [22] emulates a network using actual Linux code and enables rapid development and easy reproduction of experiments. Mininet is configured to provide a simple star topology with links running at 500Mbps and a 2ms one way delay between hosts. Iperf2 [43] clients generate the flows to one server. Each experiment was

run 30 times. The stacked bandwidth graphs show results from one representative run, and the probability distribution of results was analyzed for all results, although only a subset are shown, in violin plots due to space restrictions.

The results of typical runs are shown using stacked bar graphs of iperf bandwidth vs. time. Each bar averages bandwidth over one second, and each graph has the same scale. In these graphs, 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. Below each graph are Jain's Fairness Index [27], an index defined by the fraction of aggregate application-level throughput achieved vs. theoretical rate ($\frac{rate_{achieved}}{rate_{theoretical}}$), and an index of the 99th percentile of the Smoothed Roundtrip-time (SRTT) distribution ($\frac{RTT_{min}}{SRTT_{99th\ \%ile}}$). Note that the *latency index* inverts the theoretical:measured ratio compared to the bandwidth indices. Each index ranges from 0 to 1.0, and the combination of all three should be examined in order to evaluate the complete performance of a congestion protocol. The probability density curves showing the variation in results over 30 runs are shown after the stacked bandwidth graphs using violin plots.

5.1 Convergence

The first experiment demonstrates whether a technique can converge quickly to equal shares, maximize aggregate throughput, and minimize latencies while flows start and stop. Each iperf2 client precedes the next by five seconds and continues transmitting five seconds longer than the client that follows it. The bandwidth in each bar should be equally shared. Due to TCP overhead, 0.97 is the maximum Goodput index possible in this scenario.

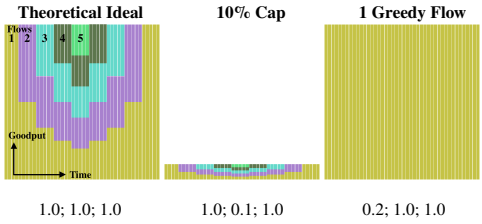

Figure 2: **Theoretical Ideal, 10% Capped, and 1 Greedy Flow examples.**
Stacked Goodput vs. Time, five converging flows. Indices below: (1) Jain's Fairness Index; (2) Goodput Index; and (3) Latency Index of the Smoothed Round-trip-time distribution. Higher is better, and 1.0 is ideal. See beginning of Section § 5 for explanations.

The ideal theoretical graph for the five converging flows is shown in Figure 2 along with a situation where flows are fair with regard to each other, but are shaped to prevent them from fully utilizing the link's available bandwidth. Finally, one greedy flow is shown consuming all available bandwidth.

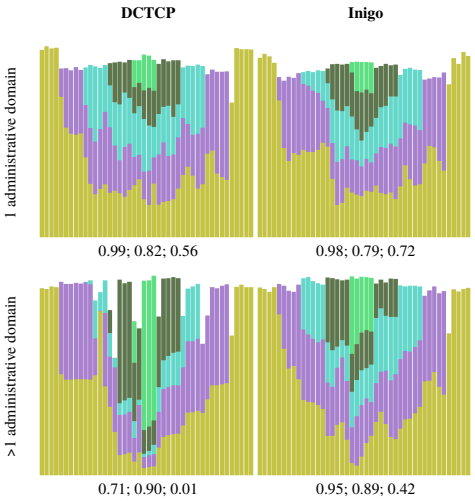

Figure 3: **Inigo improves upon DCTCP's Latency Index up to** $1.3\times$ **in simple environments and up to** $42\times$ **when not all components can be modified.**

In Figure 3, we see DCTCP and Inigo in two scenarios. Senders, receivers, and the network are all set up to cooperate in the first 1 administrative domain case, while only the sender is configured in the >1 administrative domain case. Inigo's worst case Latency Index is up to $42\times$ better the competition while its Fairness and Goodput Indices are similar. This means that Inigo is good at fully utilizing links while encouraging good buffer behavior (i.e. low occupancy and draining).

Figure 4 shows how all modes of Inigo have comparable or better performance to that of DCTCP. $\text{Inigo}_{\text{ECN}}$ would have the exact same performance profile as DCTCP if we did not include a small Slow Start modification, where the first CE marked ACK in an observation window causes an immediate exit. Our latency index emphasizes the importance of tight distributions, and $\text{Inigo}_{\text{RTT}}$ does well without any help from ECN or the receiver. Other TCPs are not shown because their goodput and 99_{th} percentile latency indices are not competitive.

5.2 Incast

Our incast experiment starts connections from eight iperf2 clients to one server simultaneously and lasts for five seconds. This type of scenario is common in parallel storage systems. While bad cases of incast involve many more clients, eight is sufficient in this Mininet experiment to show the difference in behavior between TCPs that overfill buffers and those that keep bottleneck buffer occupancy low.

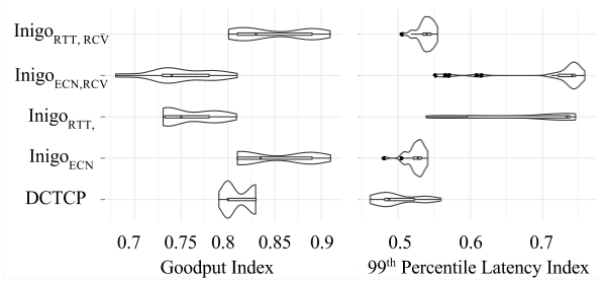

Figure 4: **Inigo's ECN mode optimizes for latency while its delay-based mode targets greater aggregate goodput while keeping tail latencies low. The Inigo receiver further decreases latencies while sacrificing little bandwidth.**
Inigo$_{RTT,RCV}$ (RTT-based sender with Inigo receiver), Inigo$_{ECN,RCV}$ (ECN-based sender with Inigo receiver), Inigo$_{ECN}$ (ECN-based Inigo sender), Inigo$_{RTT}$ (RTT-based Inigo sender), and DCTCP. Due to limited space, DCTCP's Reno fallback mode and other TCPs with smaller goodput and latency indices are not included. Probability density plot of goodput and latency indices. Right and thicker is better. Boxplot showing median, quartiles, and outliers included.

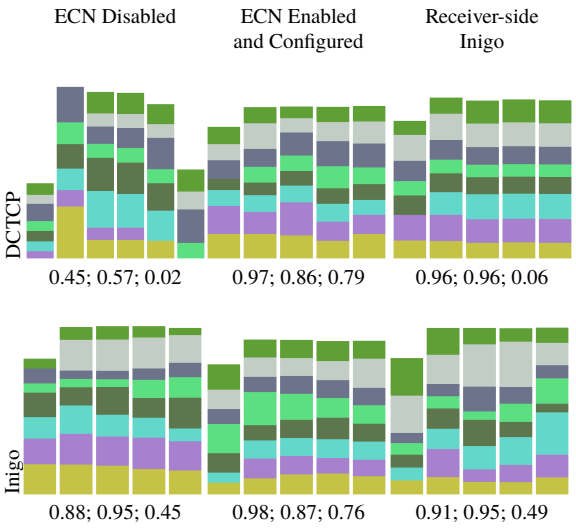

Figure 5: **Inigo's fairness, aggregate goodput, and latency indices are all significantly superior to DCTCP's Reno fallback.**
Stacked Goodput vs. Time, five converging flows. Indices below: (1) Jain's Fairness Index; (2) Normalized Goodput; and (3) Latency index of the Smoothed Round-trip-time distribution. Higher is better.

Figure 5 shows that Inigo performs well in a modest incast experiment. And the Inigo receiver can do much to improve the fairness of DCTCP's Reno fallback. CDG and CUBIC results are not shown due to space restrictions and since their performance indices are in line with those seen in the convergence experiment.

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.

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.

When Inigo's sender and receiver are used together, their latency, bandwidth, and fairness indices are up to 1.3× better than the best deployable solution. And only senders can be modified, Inigo's latency index is up to 42× better than the competition.

7 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 (including comparisons with other TCP variants), and the Mininet experiment framework can be downloaded from GitHub.

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

Acknowledgments

Many thanks to support from Los Alamos National Laboratory and the Institute for Scalable Scientific Data Management, 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] Bensley, S., Eggert, L., Thaler, D., Balasubramanian, P., and Judd, G. Datacenter tcp (dctcp): Tcp congestion control for datacenters. https://tools.ietf.org/html/draft-ietf-tcpm-dctcp-01, 2015.

[5] 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.

[6] Borman, D., Scheffenegger, R., and Jacobson, V. Rfc 7323: Tcp extensions for high performance. https://tools.ietf.org/html/rfc7323, 2014.

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

[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. Network transmit queue limits. https://lwn.net/Articles/454390/.

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

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

[14] 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.

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

[16] 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.

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

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

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

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

[21] 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.

[22] 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.

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

[24] Jacobson, V. Congestion avoidance and control. In *ACM SIGCOMM computer communication review* (1988), vol. 18, ACM, pp. 314–329.

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

[26] 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.

[27] Jain, R., Chiu, D.-M., and Hawe, W. A quantitative measure of fairness and discrimination for resource allocation in shared computer systems.

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

[29] 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.

[30] 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.

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

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

[33] 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.

[34] 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.

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

[36] 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.

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

[38] 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).

[39] 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.

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

[41] 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.

[42] 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.

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

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

[45] 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.

[46] 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.