Implementing the 'Prague Requirements' for Low Latency Low Loss Scalable Throughput (L4S)

Bob Briscoe* Koen De Schepper† Olivier Tilmans† Mirja Kühlewind‡ Joakim Misund§ ¶
Olga Albisser¶ Asad Sajjad Ahmed¶ §

Abstract

This paper is about a new approach for ultra-low queuing delay for *all* Internet traffic. 'Ultra-low queuing' means a few hundred microseconds of queuing delay on average, and about 1ms at the 99th percentile. This is 10 times better than state-of-the-art AQMs (FQ-CoDel and PIE). And all traffic means not just low rate VoIP, gaming, etc. but also capacity-seeking TCP-like applications. It is achieved by addressing the root cause of the problem—the congestion controller at the source.

The solution is to use one of the family of 'scalable' congestion controls. The talk will explain what that means, but for now it will suffice to say that Data Centre TCP (DCTCP) is an example. But there are other examples, including a scalable congestion control for real-time media. So the task has been to make it possible and safe to incrementally deploy congestion controls like DCTCP over the public Internet.

The whole architecture is called Low Latency Low Loss Scalable throughput (L4S). Although this all sounds rather grandiose, it actually only involves a few incremental changes to code in hosts and network nodes. You will recognise some of these, because most are desirable in themselves, and already in progress (e.g. Accurate ECN and RACK). However, focusing on all the parts loses sight of the sum. So this paper starts by explaining the sum of the parts - the bigger motivation for all the changes together. Then the body of the paper dives into the changes to DCTCP needed to make it safely coexist with other traffic, and to improve performance when used over the Internet—to satisfy the 'Prague L4S Requirements'. The resulting TCP implementation is called TCP Prague.

*Independent
†Nokia Bell-Labs
‡ETH Zürich
§Department of Informatics, University of Oslo
¶Simula Research Laboratory

1 Low Delay *and* High Bandwidth

Traditionally it has been believed that capacity-seeking applications (TCP-like) always build a queue so they cannot have extremely low (sub-millisecond) queuing delay. This has led to the mistaken idea that an application fundamentally cannot have both ultra-low latency and maximal throughput. With L4S every application can have both. It is ultimately intended to replace 'best efforts' as the new default Internet service.

Not only does L4S remove all the unnecessary and variable lag from today's applications (e.g. gaming, everything on the web, video chat), but it also enables tomorrow's applications that need both high bandwidth and extremely low delay. For instance, high definition video conferencing and video chat, cloud-rendered interactive video, cloud-rendered virtual reality, augmented reality, remote presence with remote control, and others yet to be invented.

The L4S architecture [9] is in the late stages of standardization on the IETF's experimental track. it involves incremental changes to hosts and network [14, 13]. Both parts of L4S are in large part integrations of existing tried and tested pieces. The aim has been to minimize risk and to ease incremental deployment but to still achieve radically better performance.

Back in summer 2015, the day after the first demonstration of applications using L4S over a DSL broadband link, 30-odd DCTCP-folks got together during the IETF in Prague and agreed a prioritized set of changes to DCTCP that would be needed to deploy it over the public Internet. The list was dubbed the 'TCP Prague requirements', which were subsequently written into an IETF spec. They have since been renamed the Prague L4S Requirements, because they also apply to UDP-based transports.

This paper introduces the important components of the first 'TCP Prague' implementation to satisfy these requirements, which has been open-sourced in Linux. As well as outlining rationale, it explains how to use it and gives the maturity of each component implementation, plus some points

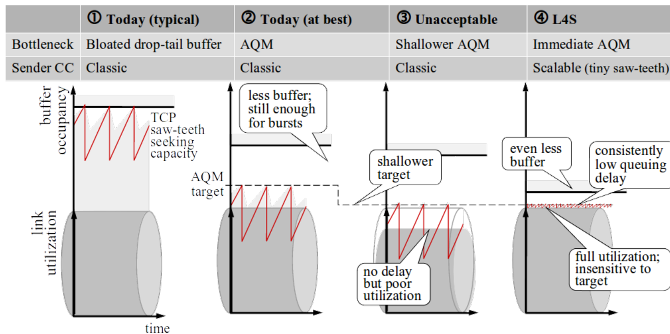

Figure 1: Scalable Congestion Control resolves the Delay-Utilization Dilemma

of interest and issues where decisions are needed. Before diving into these nuts and bolts, it starts with an overview of the sum of the parts.

2 L4S Overview

The performance gains of L4S depend on the sender using a 'scalable' congestion[1] controller. Data Centre TCP (DCTCP) is an example of a scalable congestion control. It is already available as a loadable kernel module in Linux (and in other OSs such as Windows and FreeBSD). For applications over UDP, a scalable variant of the SCReAM congestion controller has already been written for real-time media[2] and the QUIC transport protocol provides the necessary feedback mechanisms for a scalable congestion control like DCTCP to be added.

At the time of writing, end-to-end transport protocols like TCP or QUIC use a non-scalable or 'classic' congestion control by default (e.g. Cubic, Reno or BBR). This is because, until now, it has not been safe to deploy scalable congestion controls on the public Internet—they would out-compete classic traffic sharing the same bottleneck. Later (§ 2.4) we will explain how L4S solves that coexistence problem. However, first we need to explain what a scalable congestion control is, and why it gives such low delay.

2.1 Why L4S gives such Low Queuing Delay

Figure 1 visualizes link capacity as the size of a pipe and buffer capacity as the vertical space above it. The light grey shading within the buffer represents the queue and the large red saw-teeth of a classic congestion control can be seen continually seeking out available capacity in any of scenarios (1)–(3); as it alternately increases the congestion window then reduces on detecting a loss—typically by half.

[1]The term 'congestion' is used for the routine outcome of the process that seeks out capacity. It does not (necessarily) mean capacity is insufficient. Rather it is a healthy sign that applications can fully utilize capacity.

[2]https://github.com/EricssonResearch/scream

If the buffer is sized to hold more than one base round trip time (RTT) of data ('bufferbloat'), halving the window from full will still leave a standing queue (1). This is what causes the delay when small web or game transfers have to sit behind a long-running TCP transfer.

AQM (2) proactively introduces packet drops at a target of 10–20 ms of queue. This is sufficient for the RTT of the majority of flows. However, any transfers with a larger RTT cannot fully utilize the link. if the AQM target were pushed down to 1 ms, severe underutilization would result for nearly all transfers, depicted as a smaller grey pipe (3).

Scalable (L4S) congestion controls adopt a similar saw-toothing approach, except the sender scales its window reduction to the extent of recently experienced congestion. In contrast, classic congestion controls make a worst-case reduction at the first sign of the existence of any congestion. Thus, when a scalable CC's window is roughly correct, it continually makes small adjustments; shown as tiny red saw-teeth (4). Nonetheless, when congestion suddenly increases, scalable controls adapt their adjustment to the severity of congestion.

In summary, the root cause of excessively variable queuing delay is the sender's congestion control. Changes in the network can improve matters up to a limit. But, for further improvements without compromising utilization, the sender has to change—to a scalable (L4S) congestion control.

2.2 L4S: No Congestion Loss

Recall that a classic congestion control induces at least one loss at the tip of each sawtooth. So, it would seem that the significantly more frequent saw-teeth of L4S will prohibitively increase the loss level. On the contrary, L4S removes all congestion losses by mandating the use of Explicit Congestion Notification (ECN).

L4S-ECN uses the same mechanism as Classic ECN [22] where an AQM indicates congestion by setting the two ECN bits in the IP header instead of dropping packets. However, whereas a classic ECN marking has to have the same strength as a loss, an L4S-ECN marking has a much lower strength [5], which is why each reduction is much smaller. Senders set the ECN field to ECT(1) to indicate that they support the much smaller response to congestion of L4S [14]. The resulting ECN codepoints are shown in Table 1.

The mandatory use of ECN for L4S removes two further significant sources of delay:

- Loss delay: The intermittent delay detecting and repairing congestion losses, which particularly impacts short web-like data transfers.
- Feedback delay: When a classic AQM detects a burst of queuing, it holds back from discarding a packet for 100–200 ms in case the burst subsides of its own accord. The resulting bursts of delay harm applications, particularly real-time media. The root cause is that a discard-

Codepoint	Binary	Meaning
Not-ECT	00	Not ECN-Capable Transport
ECT(0)	10	ECN-Capable Transport (Classic)
ECT(1)	01	ECN-Capable Transport (L4S)
CE	11	Congestion Experienced

Table 1: The ECN Field of the IP header (v4 or v6) for the L4S Experiment

based AQM faces a dilemma between two impairments: a spike of delay vs. perhaps unnecessarily discarding a packet. By mandating ECN, L4S escapes this dilemma. So, it can signal queue growth without delay thus removing the delay spikes that Classic AQMs suffer from. With L4S, any smoothing of signals is applied at the sender at the timescale of its own RTT, not in the network, which doesn't know each flow's RTT.

2.3 Why L4S is Scalable

The key to the scalability of scalable congestion controls like DCTCP and TCP Prague is that the number of ECN marks they induce per round trip (2 on average) remains invariant as flow rate scales.

In 2008 TCP Cubic was introduced because TCP Reno had reached its scaling limit. TCP Cubic is now reaching its scaling limit. Every time Cubic takes a loss and reduces its window, it takes hundreds of RTTs to refill the buffer enough to induce the next loss, during which time it is running blind to any newly available capacity. And the faster Cubic transfers data, the longer it runs blind between loss feedback signals. For instance, at 100Mb/s Cubic induces a loss every 250 round trips. And an 8-fold rate increase doubles the duration of each sawtooth to 500 round trips.

While we use unscalable classic controllers like Reno or Cubic, many applications (e.g. web browsers, speedtest.net) open multiple data flows in order to fully utilize capacity. TCP Prague and the family of scalable congestion controls will always be able to fill high bandwidth and/or high RTT links with a single flow.

2.4 Coexistence of L4S with Existing Traffic

Until now, to reap the benefits of scalable congestion controls like DCTCP on the public Internet, all devices on the Internet would have had to switch over at the same time. That's because the frequent ECN-marking seems like heavy congestion, so it fools classic congestion controls into starving themselves. Indeed, DCTCP was so named because it was originally expected to only be applicable in private data centres, where the administrator could arrange such a switch-over.

The IETF's Dual Queue Coupled AQM framework [13] is the key piece of the L4S architecture that solves this coexistence problem. A reference Linux qdisc implementation was open-sourced in July 2016 and it has recently been refactored as DualPI2 [2].

In most networks the access link is designed to be the bottleneck. So most of the benefit can be gained by deploying this queue structure at the ingress to the access link, preferably in both directions. The cable industry recently made the DualQ Coupled AQM mandatory for DOCSIS 3.1 [1] Cable Modems (CMs) (upstream) and CM Termination Systems (downstream). This structure is also being implemented in high speed switch chip-sets. So it is becoming likely that this queue structure will be common at access bottlenecks.

The DualQ Coupled AQM consists of two queues, one called LL for Low Latency, and one called Classic. Each has its own AQM. ECT(1) packets are classified into the LL queue.

If a packet arrives at an L4S AQM already marked CE, the AQM cannot tell whether it was originally ECT(0) or ECT(1). Nonetheless the IETF has decided that it is safe to (mis-)classify CE packets as L4S even though they could be reordered if they were originally ECT(0), because a few packets delivered early will not cause spurious retransmissions.

L4S traffic induces a very shallow queue, which is separated from the larger queuing delay induced by classic congestion controls. Nonetheless, from a bandwidth perspective it makes the two queues seem to be one—even though the two queues feed into a priority scheduler, this is only for latency priority, there is no bandwidth priority. For instance, if there are n flows in the L4S queue and m flows in the classic queue, each flow will get roughly $1/(n+m)$ of the aggregate capacity available to both queues. The network doesn't identify any flows in order to do this — it even works if some or all of the flow identifiers are encrypted.

It works by coupling the congestion signalling level (drop or ECN-marking) from the classic queue to the LL queue so that the congestion level fed back to senders from the LL queue appears as if the classic flows are in the LL queue. Then L4S flows leave enough space between their packets for the classic traffic.

The coupling also takes account of the different way the two types of congestion controllers reduce their rate in response to congestion signals. it is well-known (by TCP researchers) that the packet rate r of a classic TCP flow responds to loss level p by following the inverse square-root law shown in Figure 2. In contrast, a scalable congestion controller follows an inverse linear law (which is why it is scalable and classic TCP is not).

The coupling is arranged so that the classic queue sees the square of the congestion level coupled across to the L4S queue. This counterbalances the square root in the source's response formula. So the upshot is that all flows share out

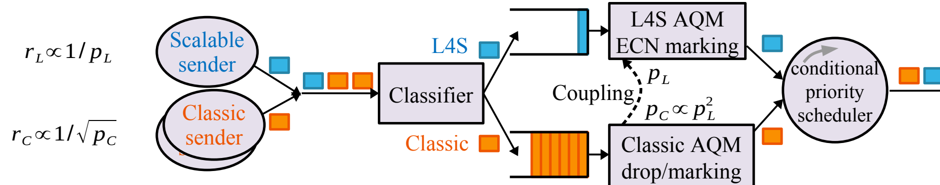

Figure 2: DualQ Coupled AQM; r: packet rate per flow, p: drop or marking probability

the capacity equally. This is the theory, and it also works amazingly well in practice. Further details about the DualQ Coupled AQM are in the complementary paper on the Linux implementation of the DualPI2 qdisc [2].

We have seen that, as sources upgrade to L4S, the coupled AQM ensures they safely coexist with Classic traffic. But what if the upgrades happen the other way round—a source upgrades to L4S but there is no coupled AQM at the bottleneck? This is also safe because the host can detect that the current bottleneck link does not support L4S. Then its congestion control has to fall back to classic behaviour, so as not to starve competing classic flows. This is covered in § 3.

In summary, the lower strength L4S-ECN marking is central to all three aspects of L4S: i) **Low Latency**: the marking acts as the classifier for a separate shallow queue; ii) **Low Loss**: ECN avoids congestion loss, which in turn removes the delay of loss repair and allow immediate congestion signalling; iii) **Scalable throughput**: The frequency of L4S-ECN signalling remains invariant with flow-rate, ensuring that throughput should scale indefinitely.

3 TCP Prague

The beneficial properties of L4S traffic (low queuing delay, etc.) depend on all L4S sources satisfying a set of conditions called the Prague L4S Requirements. The name is after an ad hoc meeting of about thirty people co-located with the IETF in Prague in July 2015. The meeting was hastily arranged the day after the first public demonstration of an interactive cloud-rendered video application running over an Internet access link with L4S support.

The meeting agreed a list of modifications to DCTCP to focus activity on a variant that would be safe to use over the public Internet. it was suggested that this could be called TCP Prague to distinguish it from DCTCP.

As explained above, DCTCP already implements a scalable congestion control. So most of the changes to make it usable over the Internet seemed trivial, often merely involving adoption of other parallel developments like Accurate ECN TCP feedback or RACK. Some were more challenging (e.g. RTT-independence). And others that seemed trivial became challenging (e.g. when confronted with the complex set of bugs and behaviours that characterize today's Internet).

The rest of this paper gives the rationale for each modification to DCTCP, its current status, and points of interest. Configuration advice is included for those components that are still deemed too immature to enable by default. The sections below reflect the latest position since the IETF adopted the list of L4S requirements and documented them in more detail [14], including adding one new item (RACK). The modifications are categorized into mandatory requirements (§ 3.1) and optional performance optimizations (§ 3.2) as follows:

Mandatory Requirements

- L4S-ECN packet identification: ECT(1)
- Accurate ECN feedback
- Fall-back to Reno-friendly on Loss
- Fall-back to Reno-friendly on classic ECN bottleneck
- Reduce RTT dependence
- Scale down to fractional window
- Detecting loss in units of time

Optional Performance Optimizations

- ECN-capable TCP control packets
- Faster flow start
- Faster than additive increase

Note that the Prague L4S Requirements are not TCP-specific; they apply irrespective of wire-protocol (TCP, QUIC, RTP, SCTP, etc), and there are L4S implementations for other transports, e.g. L4S SCReAM for real-time media[3]. Nonetheless, this section focuses on how the Prague requirements have been addressed for the TCP protocol — for TCP Prague.

TCP Prague is open sourced under GPLv2 and is being developed as a congestion control module against the latest Linux net tree[4], and will be incrementally submitted to mainline. It can be loaded and enabled as follows:

```
sudo modprobe tcp_prague
sudo sysctl -w net.ipv4.tcp_congestion_control=prague
```

3.1 The Prague L4S Requirements

We begin by detailing the requirements for a transport protocol to be L4S-capable. These include both safety behavior, which prevent L4S flows from starving classic ones, and scalability requirements, which ensure that the rate of

[3] https://github.com/EricssonResearch/scream
[4] https://github.com/L4STeam/tcp-prague

the congestion signals per round trip scales together with bandwidth-delay product (BDP or window) changes.

3.1.1 Packet Identification: L4S-ECN

Satisfying the mandatory Prague L4S Requirements entitles the source to identify that its packets support L4S in the 2-bit ECN field of the IP header.

The IETF is standardizing the experimental ECT(1) codepoint (0b01) to mean L4S on the public Internet [14].

The default should be sufficient but, in a controlled environment like a data centre, the operator might have reason for TCP Prague to use ECT(0) for L4S traffic (e.g. to avoid some other local use of ECT(1)). Such a non-default scenario can be configured (for both IPv4 & IPv6) as follows:

```
sudo modprobe tcp_prague prague_ect=b
```

where the values for b have the meanings tabulated below.

b		TCP Prague	DCTCP
0	Use ECT(0)		default
1	Use ECT(1)	default	

Conversely, a DC operator might want DCTCP to use ECT(1). Therefore DCTCP could add a similar module option, but it would have to be dependent on the patch fixing DCTCP's response to loss being accepted (and not disabled via another module option). Assuming it would be similar to the TCP Prague module option (except with 'prague' replaced by 'dctcp'), we have given it a column in the table above to show the likely relationship between the defaults.

3.1.2 Accurate ECN Feedback

Unfortunately, when ECN feedback was originally added to TCP in RFC3168, it only reported at most one congestion event per round trip [22]. The rationale was that TCP Reno was required to respond to congestion no more than once per round trip. When a CE-marked packet arrives at a Classic ECN (RFC3168) receiver, in case an ACK is lost, it just solidly feeds back the Echo Congestion Experienced (ECE) flag in the TCP header (Figure 3) of every packet until the sender acknowledges it has heard by setting the Congestion Window Reduced (CWR) TCP header flag.

0–3	4–6	7	8	9	10	11	12	13	14	15
Header Length	Reserved	**AE**	**CWR**	**ECE**	URG	ACK	PSH	RST	SYN	FIN

Figure 3: The TCP header flags, including the experimental Accurate ECN (AE) flag

TCP Prague, like DCTCP or any scalable congestion controller, achieves its low queuing delay by making continual fine adjustments in response to marginal changes in the congestion level. So a TCP Prague sender has to know how many bytes arrived at the receiver in packets with their ECN field marked as 'Congestion Experienced' (0b11).

Therefore TCP Prague uses Accurate ECN (AccECN) TCP feedback, which is the IETF's experimental change to the TCP wire protocol to feed back the extent of ECN feedback [8]. Briefly, AccECN uses the three TCP header flags AE, CWR & ECE (Figure 3) to negotiate support for AccECN during the 3-way handshake. Then it re-purposes them as a 3-bit counter (called the ACE field) for each end to repeat the number of CE marks it has seen to the other.

So why doesn't TCP Prague use DCTCP's feedback scheme? DCTCP corrected the deficiency of TCP's ECN feedback with its own proprietary modification to the TCP wire protocol. However, it does not provide a way for one DCTCP peer to check that the other understands DCTCP feedback (consistent configuration of all hosts is arranged by the sysadmin in data centres). Also, DCTCP's feedback metric becomes confused when ACKs are lost. So the IETF did not consider the DCTCP scheme robust enough for the general Internet and selected AccECN instead [18].

TCP Prague uses Külewind's implementation of AccECN for Linux [17]. AccECN updates the base TCP stack and is intended to be generic for any congestion control module, because it is backward compatible with congestion controls that only need feedback of one congestion event per round trip. The AccECN code is targeted at the base TCP stack At the time of writing the AccECN code has been ported to v5.1 and an RFC submitted .

Although AccECN does not require TCP Prague, the reverse is not true. A TCP Prague sender requires both ends to have negotiated support for AccECN. Therefore, it would be perfectly valid for one TCP peer to support L4S congestion control and set ECT(1) on its packets, while the other end used a different (non-L4S) congestion control. However, this scenario could only arise if both ends supported AccECN.

The requirement that neither host in a connection can use TCP Prague unless both ends support AccECN feedback makes TCP Prague deployment rather unrewarding while AccECN is not widely supported. Therefore, we have provided a non-default sysctl option to AccECN in the TCP stack. It can be enabled during testing to force the remote peer to echo CE markings individually.

```
sudo sysctl -w net.ipv4.tcp_force_peer_unreliable_ece=1
```

It only becomes effective on connections where AccECN feedback has not been successfully negotiated but classic RFC3168 ECN feedback has. This is *not recommended* for use over the public Internet because, like DCTCP feedback, the individual echoes are not delivered reliably. Therefore it might be possible for congestion on the reverse path to degrade congestion feedback on the forward path — clearly a potentially delicate scenario. However, in controlled test environments this option is likely to prove invaluable.

This feedback mode works by setting the CWR TCP header flag proactively and continuously (meaning Congestion Window Reduced Figure 3). Normally CWR is only set once to acknowledge and stop each volley of ECE fed back from the remote peer (meaning Echo Congestion Experienced).

There was a bug in the ECN code of all early derivatives of BSD, that we have to be careful not to trigger on those remote peers that still suffer from it. If a packet with CWR in the TCP header happens to also be marked as CE in the IP header, a bugged receiver sends no feedback at all. Therefore, before our TCP Prague host sets CWR continuously, if this module option is enabled, the TCP Prague host sets both CWR and CE on its first data packet. If the feedback for that packet does not carry ECE, it knows the remote peer has the bug. Otherwise, it can safely send CWR for the rest of the connection.

3.1.3 Fallback on Loss

DCTCP includes a non-default module option to fall back to Reno behaviour on loss. However, even with the option enabled, the code only handles losses detected by a retransmission timeout not via fast retransmit. Also, on a retransmission timeout, it causes an unnecessarily large cwnd reduction by setting DCTCP's congestion level estimate (alpha) to its maximum, which impacts high BDP flows particularly badly. This was a long-standing bug that we have recently fixed for TCP Prague, and submitted a patch for DCTCP.

The original code halved `ssthresh` on a retransmission timeout, which is also the decrease used in the bugfix. Nonetheless, this raises the question of whether the reduction on loss should be less severe if TCP is in the CWR (Congestion Window Reduced) state. This would imply that there has been less than a round trip since a previous reduction due to any of i) ECN marking; ii) local qdisc congestion; or iii) TLP (tail loss probe) loss detection. In the worst case, if this earlier reduction had been a full halving, compounding another halving due to loss would result in a reduction to 1/4. Section 3.5 of the DCTCP RFC [4] gives requirements for this case, but they are ambiguous.

Firstly, we propose that it is unnecessary to adjust for exactly how much of the first round trip had passed before the loss starts a new round trip of reduction. One policy would then be to contrive the second reduction so that the compound of both reductions amounts to a reduction of 1/2. The first reduction in `ssthresh` due to ECN would have been (1-alpha/2) [in floating point terms] or (2-alpha)/2. To compound this into 1/2 requires the second reduction (on the loss) to be 1/(2-alpha) to cancel out the numerator. However, this would involve division between two u32s.

Instead, on loss we propose to reduce `ssthresh` by (2+alpha)/4 if in CWR state. This is both cheaper to process and gives roughly the desired outcome. If alpha is small, it makes the compounded reduction closely approximate to 1/2. And if alpha takes its maximum value, it results in a compounded reduction of 3/8, which is conveniently half-way between 1/2 and 1/4.

3.1.4 Fallback on Classic ECN

The requirement here is for TCP Prague to detect if it is getting ECN marking from the network that is from an AQM applying 'classic' ECN not L4S ECN marking. Otherwise, if not protected by a per-flow queuing scheduler (FQ), TCP Prague would dominate other 'classic' TCP-friendly flows, which respond much more strongly to ECN than L4S sources.

To detect this case, the sender has to monitor the smoothed RTT from the start of the connection. If SRTT increases by more than about 2 or 3 ms, it is likely to be a Classic ECN bottleneck. Then TCP Prague should respond to ECN marks once per RTT as it would for loss (§ subsubsection 3.1.3). The question of how quickly a flow could detect this scenario, and whether it would make any difference for short flows anyway is to be determined.

This requirement is yet to be coded, on the basis that Classic ECN AQMs never materialized on the public Internet until FQ-CoDel was deployed. And FQ-CoDel [16] would prevent TCP Prague starving other flows anyway. If other classic ECN AQMs appear, we will obviously have to revisit this decision.

3.1.5 Reducing RTT-Dependence

Given TCP's rate has always been inversely proportional to RTT, it might seem that this is an unnecessary aspirational requirement. However, until now the effect has always been masked by large buffers, as the following example reveals.

Imagine two competing flows over two paths with base round trips of 100 ms and 5 ms. You would think the ratio between their rates would be 20:1 so the longer RTT would starve. However, if they both experience a common (bloated) queuing delay of 95 ms the ratio of their RTTs will be far less problematic - closer to 2:1 (=(100+95)/(5+95) = 195/100).

We first tripped up on this starvation problem when we tested the DualQ Coupled AQM with DCTCP. At first we thought it was a bug in the coupling. However, it was DCTCP behaving as intended—it was just that no-one had seen the problem before, because no queues had been that small.

RTT dependence is a more significant problem for TCP Prague, given it is targeted at the public Internet where the range of RTTs is much wider than in a DC. We have implemented solutions in a simulator and chosen the best (the candidate designs and the currently chosen winner are in [7]). However, as of the time of publication it is yet to be implemented in Linux.

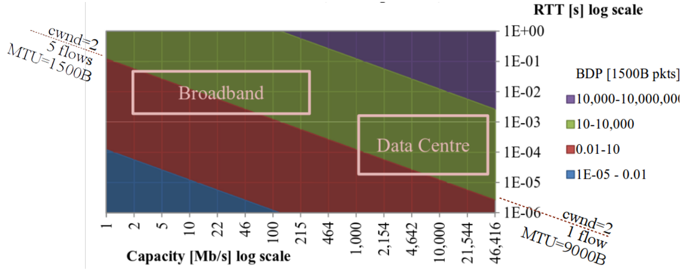

Figure 4: The need for a fractional window, to preserve a tiny queue

3.1.6 Scaling Down to Fractional Windows

While testing the coupled AQM with different numbers of DCTCP flows, we noticed that the queue started to grow beyond the AQM's delay target once a few flows were added, but only in low base RTT scenarios. We worked out that the cause was TCP's minimum window of 2 segments. With such a short queue, the RTT was so small that the flows could fill the link with less than 2 segments per RTT. Because TCP does not allow a window below 2 segments, it became unresponsive to the ECN marks from the AQM and just forced the queue to grow until the RTT was large enough to fit 2 segments per flow.

Figure 4 (adapted from [6]) shows the potential scope of this problem once queues are small. It shows ranges of bandwidth-delay product across the two dimensions of capacity and RTT. The boundary between red and green represents a BDP of ten 1500 B packets. As shown, this BDP of 10 could consist of 5 flows at their min cwnd of two 1500 B packets, or just less than one flow using larger 9000 B packets, also at its min cwnd of two. Two rectangles are overlaid to visualize the approximate scope of broadband and DC networks. Then the red-green boundary shows that a good proportion of regular networks will need to operate with a BDP of less than 10 under not atypical circumstances.

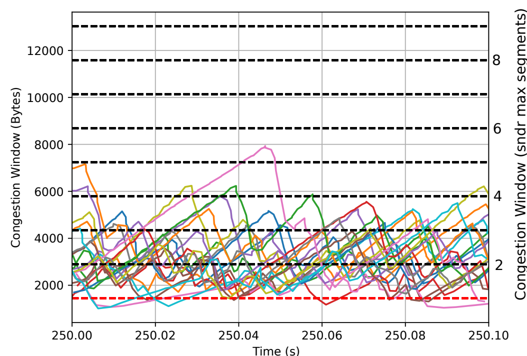

Figure 5: The congestion windows of 12 parallel TCP flows modified to support a fractional window

To change such a fundamental part of TCP has proved challenging, both in design and implementation terms. Our goal is to modify the base TCP stack for all congestion controls. Figure 5 shows cwnd below 2 SMSS with the Reno congestion control module loaded.[5] However, at the time of writing we are still in the process of debugging synchronization problems when using DCTCP.

There were two challenges to overcome in the base TCP stack and two in the TCP Prague CC module:

- A Linux sender counts the window in whole max-sized segments (SMSS). We have added a variable to hold a fractional part of the window (separate from `snd_cwnd_cnt`).

- Packet conservation cannot rely on the ACK clock when the required spacing between packets is greater than the time an ACK takes to return. Instead we used the pacing facilities now in the Linux kernel.

- With a fractional cwnd, multiplicative decrease scales well, but the constant additive increase of 1 SMSS per RTT keeps pushing cwnd above 1 SMSS no matter how small it is after a decrease. Therefore, we have replaced the constant additive increase with a variable, and we are experimenting with recalibrating it on each decrease of ssthresh to be

  ```
  add =  ADD0 * lg(ssthresh/SSTHRESH0 + 1);
  ```

  where the constants (ADD0 = 256 B; SSTHRESH0 = 512 B) are integer powers of 2 arranged so that the increase is roughly 1 SMSS for the current typical range of ssthresh, which ensures rough interoperability with existing TCP.

- The exponentially weighted moving average (EWMA) algorithm of DCTCP is meant to run every round trip clocked by ACKs. However, ACKs arrive less often than round trips. Therefore, in the fractional window regime we run the EWMA on each ACK and algorithmically adjust it, as if it ran every whole round trip.

We have not included these modifications in the TCP Prague implementation. Once we have a solution that tests well in a range of scenarios, we will make it available separately for researchers to evaluate. We encourage others to come up with alternatives. Once we have more confidence in the stability of a solution we will add it to the TCP Prague implementation, probably initially as a default-off systctl option.

[5] Testbed config: 20 TCP Reno flows; bottleneck link rate: 200 Mb/s; base RTT: 300 μs; SMSS: 1448 B; AQM: RED 0–10% over 1–3 ms;

3.1.7 Detecting Loss in Units of Time

RACK [10, 11] is a sender-only change to the bases TCP stack that detects loss by counting in units of round trip times instead of counting packets (as in the traditional 3 DupACK rule).

It occurred to us that, as well as simplifying the TCP implementation, counting in time also has the potential to enable link designs to scale better in future. As flow rates have increased over the years, the need to keep reordering within 3 packets has required links to hold reordering within an increasingly tight timespan (because it takes less time to transmit 3 packets). Once senders count in time units to detect loss, the reordering constraint on links no longer tightens as flow rates scale up.

However, this opportunity can only be realized if a link is certain that all traffic comes from senders that are using a loss-detection algorithm that counts in time units. We realized that we could exploit this opportunity in L4S links now by making it mandatory for L4S senders to detect loss in units of time.

Superficially it seems that implementing this requirement is just a case of using RACK, which has been the default in Linux kernels since v4.18. There is just a slight wrinkle in that RACK bootstraps itself with the 3 DupACK rule, until it has a stable measurement of the path's reordering degree. Unless we eliminate all packet counting for loss detection (including at bootstrap) in L4S transports the scaling advantage of RACK will be lost.

As of the time of writing, TCP Prague uses RACK as-is. We (and the developers of RACK) are considering the best way to modify RACK's bootstrap to allow links to scale, such as:

- Bootstrap RACK with a reordering window of, say, SRTT/8, where the value of the SRTT is informed by the RTT during the 3WHS as well as any other cached information, such as the per-destination cache or a TFO cookie [12].
- Use the 3 DupACK rule, but never send the packets of the initial window back-to-back. Instead, ensure they are paced over the RTT so that packet spacing will stay constant as flow rates scale.

It is suspected that inaccuracy in the early RTT measurement will sometimes cause spurious retransmissions or take too long to detect loss. Therefore the second approach is currently preferred. The choice for TCP Prague depends on the flow-start approach adopted (see § 3.2.2).

3.2 TCP Prague Performance Optimizations

3.2.1 ECN-Capable TCP Control Packets

When ECN was originally added to TCP [22], the ECN capability was precluded from TCP control packets and retransmissions. The concerns that led to that decision have since been rebutted and the IETF is proposing an experimental track scheme called ECN++ that allows all TCP control packets and retransmissions to be ECN-capable [3].

Loss of a control packet tends to impact performance much more than loss of a data packet. In particular, loss of a SYN leads to a 1 s timeout. Therefore protecting control packets with ECN significantly improves performance.

The IETF's ECN++ spec. requires that the ECN capability on SYNs and pure ACKs is conditional on negotiating Accurate ECN feedback. Fortunately AccECN is also mandatory for TCP Prague. Therefore, all TCP Prague packets are ECN-capable. The DCTCP CC module already requires ECN to be enabled on all packets, so TCP Prague merely copies that.

Using ECN-capable control packets over the Internet is not as straightforward as in a DC, because of the possibility that middleboxes and/or servers reject packets that do not conform to earlier IETF specs. TCP Prague does not yet implement all the fall-back mechanisms suggested in [3].

The Over-Strict ECN Negotiation Bug Back in 2012, there was some concern that the IP-ECN field might be mangled as it traversed Internet paths. Section 6.1.1 of the ECN RFC [22] says "A host MUST NOT set ECT on SYN ... packets", So it was assumed that a SYN with a non-zero ECN field would always be a symptom of network mangling. Therefore, on arrival of a SYN attempting to negotiate ECN at the TCP layer, a check for a non-zero IP-ECN field (called 'strict ECN negotiation') was added to Linux. If the IP-ECN field on the SYN was non-zero, ECN was disabled for the rest of the connection so that a potentially mangled ECN field would not be relied on.

In an RFC "MUST NOT" does not mean "MUST never". Inevitably, RFC 3168 has been updated by RFC 8311 [5] to allow ECN-capable control packets. However, Linux's strict ECN negotiation is now so widespread that it has become pointless to try to send an ECN-capable SYN—you will nearly always just disable ECN completely. Indeed, according to measurements in a Nov 2017 study, of the 82% of the Alexa top 50k web servers that supported ECN, 84% disabled ECN in response to ECT or CE on a SYN [19].

There is a way out of this impasse though. The strict-ecn test can be made specific to an RFC3168 ECN set-up SYN. As well as solely testing that the two ECN flags are set (CWR and ECE at bits 8&9 in Figure 3), the fix also checks that the first four TCP header flags (res1 at bits 4-7) are zero. This makes the test specific to clients not using any of the 4 reserved flags, which gives the client a way to exclude itself from the test. Indeed, the IETF's specification for making SYNs ECN-capable requires the 4th flag to be set to 1. So, when sending to a patched server, the client's SYN automatically excludes itself from the strict ECN negotiation test. Therefore the client can send ECN-capable SYNs without

triggering the server to disable ECN.

We have submitted a patch that uses this idea, and made it is easy to back-port in an attempt to fix the over-strict ECN negotiation 'bug' on as large a proportion of the deployed base of Linux servers as possible.

3.2.2 Faster Flow Start

DCTCP uses the traditional slow start algorithm, which means it exits slow start at the first ECN feedback. As we have explained, DCTCP induces significantly higher ECN-marking frequency than a classic TCP algorithm with ECN support, such as Cubic [23]. So, it is already well-known from DC experience that a DCTCP flow trying to push into other DCTCP traffic at the bottleneck is likely to be knocked out of slow start early.

TCP Prague has the same problem, but compounded by another. On the Internet, unlike in a DC, the bottleneck link tends to be one or two orders of magnitude slower than the sending NIC. So it is very easy for a TCP Prague sender to tip the bottleneck over its shallow ECN threshold during slow start, even if no other traffic is present.

Either or both these effects tend to cause a new flow to take a long time to reach its intended rate. Thus, both DCTCP and TCP Prague are weak at getting up to speed. Nonetheless, on the plus side, they only induce a tiny queue.

The 'obvious' solution seems to be to make TCP Prague's slow start respond to the extent of ECN marking, not just the existence of a single mark, which is the essence of DCTCP's approach in the congestion avoidance phase. However, this is problematic, because TCP Prague cannot immediately know whether ECN markings are coming from a classic ECN AQM or an L4S-ECN AQM. it can detect the difference once it's had some time to assess the evolution of the RTT (see § 3.1.4). But it cannot sit back and watch for a number of round trips in order to decide in hindsight what it should have done on the first ECN mark.

Another 'obvious' solution seems to be to add support for a special acceleration signalling protocol to all L4S AQMs and to all L4S congestion controls. However, all special acceleration signalling schemes fail on incremental deployment grounds. During flow start-up, it is very common for the bottleneck to move. This is because another TCP flow (or flows) that was filling a high rate link (e.g. the 800 Mb/s link in Figure 6) will rapidly decrease its rate to let in a new flow. So what appeared to be hardly any available capacity suddenly jumps (to 400 Mb/s in the example). Then, if the red node in the example emits a new acceleration signal, the source could easily overrun the 100 Mb/s link where the new signal is not supported.

Based on these arguments, for flow start we believe TCP Prague will have to use a signal that is universally supported, i.e. loss or delay. The most promising current approach is paced chirping [21], which uses delay.

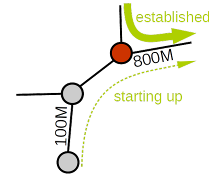

Figure 6: The problem of deploying new acceleration signals

Up to now, the most aggressive flow-start has tended to cause the worst queue overshoot. However, the flow completion times of paced chirping compare with the best (Cubic with or without hystart [15]) while its queuing compares with the low single digit milliseconds of the least aggressive approach (DCTCP).

Chirps are sequences of one or two dozen packets with increasingly reducing gaps between them. The sender analyzes the ACK-spacing to determine the available capacity from the point at which the gaps between the ACKs diverge from the gaps between the sent packets. Paced chirping involves sending out a sequence of chirps separated by guard intervals. Then as the ACKs arrive, the sender averages the estimates of available capacity to determine a better average rate for each chirp. And the less variance there is between the estimates, the more rapidly it closes up the guard intervals.

Although promising, paced chirping is still considered research. So it is not (yet) included in the TCP Prague implementation. It is under continual development, but the Linux implementation is open-sourced[6] so it can be manually added to TCP Prague (or to any other congestion control). An overview of the Linux implementation is available in the Netdev 0x13 paper about it [20].

3.2.3 Faster than Additive Increase

DCTCP uses the unmodified additive increase of TCP Reno. At a window increase of 1 segment per round trip, this makes DCTCP slow to pick up newly available capacity. TCP Cubic [23] was developed to solve this problem. However, it is now reaching its own scaling limits. For instance, over a 20ms RTT, at 100 Mb/s, each Cubic sawtooth is 250 round trips long and this duration doubles every 8× increase in flow-rate.

The two can be compared in Figure 7 (copied from [20]), which shows a simulation of an increase in capacity from 50 Mb/s to 100 Mb/s, for instance, due to a flow finishing. The plots show how fast different congestion controls (separately) pick up the newly available capacity. The base RTT is 100 ms.

The awful responsiveness of unmodified DCTCP is most apparent, taking 600 round trips (Reno would be no different). Cubic takes about 120 round trips. Nonetheless, it shows that DCTCP can be modified to take just 6 round trips

[6] https://github.com/JoakimMisund/PacedChirping

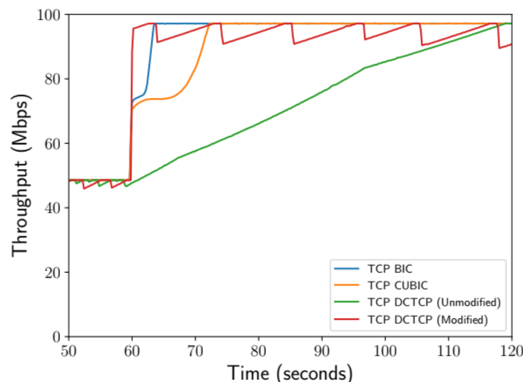

Figure 7: Comparison of Paced Chirping with other approaches to regain newly freed up capacity, each simulated separately

to get within 95% of full utilization, in addition to the number of quiet RTTs that it is configured to wait over before probing (e.g. 2).

The modified DCTCP still uses Reno-like additive increase while in congestion avoidance. But after a couple of rounds with no ECN signals it deems that it can start probing for available capacity. It starts introducing a few paced chirps, which rapidly measure the newly available capacity in the subsequent rounds, then it rapidly increases to fill the target capacity.

Not only is the modified DCTCP much faster but the overshoot of the queue with paced chirps is tiny—-the same as unmodified DCTCP (about 1 ms). In contrast, even though they increase more much slowly, the queue delay overshoot of Cubic (and BIC) fills the buffer.

Cubic and other classic controls do not have the potential to be modified like this. Cubic expects a signal (loss or ECN) every few hundred round trips on average. When capacity opens up, it takes a few hundred more rounds before Cubic can even start to think that the signals might have stopped. In contrast, DCTCP can rapidly tell when 2 marks per round has stopped. And DCTCP's scaling property (§ 2) mean its superior responsiveness will remain invariant while others will degrade as flow rate scales.

We have tried adding continual chirps to Reno during congestion avoidance, but this adds a lot of delay noise, which makes it hard and therefore slow for flows entering the system to measure available capacity using chirps.

Paced chirping can be added to any congestion control during slow start (including re-starts). But it is only applicable to scalable controls, like DCTCP and TCP Prague, during congestion avoidance.

In both cases, paced chirping needs more research before it can be considered mature enough to ship with TCP Prague or DCTCP.

Linux code: none | none (simulated) | research private | research opened | RFC | mainline

Requirements	base TCP	DCTCP	TCP Prague
L4S-ECN Packet Identification: ECT(1)		module option	mod opt default
Accurate ECN TCP feedback	sysctl option	?	mandatory
Reno-friendly on loss		inherent?	inherent
Reno-friendly if classic ECN bottleneck			TBA if nec.
Reduce RTT dependence			simulated
Scale down to fractional window	in progress		
Detecting loss in units of time	default RACK	default RACK	mandatory
Optimizations			
ECN-capable TCP control packets	sysctl option off	default on	default off→on later
Faster flow start	in progress		
Faster than additive increase		in progress	

Figure 8: TCP Prague Implementation Status

4 Summary

Like DCTCP, L4S derives its extremely low latency from a complementary interaction between the sender's congestion control and a very simple AQM in the network. This paper has introduced the whole L4S architecture. But its focus is the transport protocol, in particular TCP Prague, which is largely where the interesting parts are.

Figure 8 summarizes the Linux implementation status of the alterations to DCTCP to turn it into TCP Prague, by addressing each Prague L4S Requirement or Optimization. It can be seen that half the components are either already optional parts of the base TCP stack or they are works in progress motivated for other reasons than just L4S—TCP Prague is essentially a configuration that makes these options mandatory. Three of the remaining parts are also applicable to DCTCP.

That leaves i) reduced RTT dependence and ii) Classic ECN detection. As already detailed, the former has been designed and simulated, but not yet implemented. And the latter is thought to be unnecessary for the current Internet, but can be added if it proves necessary.

5 Acknowledgments

Mirja Kühlewind was partly supported by the European Commission under Horizon 2020 grant agreement no. 688421 Measurement and Architecture for a Middleboxed Internet (MAMI), and by the Swiss State Secretariat for Education, Research, and Innovation under contract no. 15.0268. The initial work on L4S was part-funded by the European Community under its Seventh Framework Programme through the Reducing Internet Transport Latency (RITE) project (ICT-317700). In neither case does support imply endorsement.

Given TCP Prague is an integration of many people's work, it would be infeasible to acknowledge everyone involved. Therefore the acknowledgements in their work referenced here will have to suffice.

References

[1] Data-Over-Cable Service Interface Specifications DOCSIS® 3.1; MAC and Upper Layer Protocols Interface Specification. Specification CM-SP-MULPIv3.1-I17-190121, CableLabs, Jan. 2019.

[2] ALBISSER, O., DE SCHEPPER, K., BRISCOE, B., TILMANS, O., AND STEEN, H. DUALPI2 - Low Latency, Low Loss and Scalable (L4S) AQM. In *Proc. Netdev 0x13* (Mar. 2019).

[3] BAGNULO, M., AND BRISCOE, B. ECN++: Adding Explicit Congestion Notification (ECN) to TCP Control Packets. Internet Draft draft-ietf-tcpm-generalized-ecn-03, Internet Engineering Task Force, Oct. 2018. (Work in Progress).

[4] BENSLEY, S., THALER, D., BALASUBRAMANIAN, P., EGGERT, L., AND JUDD, G. Data Center TCP (DCTCP): TCP Congestion Control for Data Centers. Request for Comments RFC8257, RFC Editor, Oct. 2017.

[5] BLACK, D. Relaxing Restrictions on Explicit Congestion Notification (ECN) Experimentation. Request for Comments RFC8311, RFC Editor, Jan. 2018.

[6] BRISCOE, B., AND DE SCHEPPER, K. Scaling TCP's Congestion Window for Small Round Trip Times. Technical report TR-TUB8-2015-002, BT, May 2015.

[7] BRISCOE, B., AND DE SCHEPPER, K. Resolving Tensions between Congestion Control Scaling Requirements. Technical Report TR-CS-2016-001, Simula, July 2017.

[8] BRISCOE, B., KÜHLEWIND, M., AND SCHEFFENEGGER, R. More Accurate ECN Feedback in TCP. Internet Draft draft-ietf-tcpm-accurate-ecn-07, Internet Engineering Task Force, July 2018. (Work in Progress).

[9] BRISCOE (ED.), B., DE SCHEPPER, K., AND BAGNULO, M. Low Latency, Low Loss, Scalable Throughput (L4S) Internet Service: Architecture. Internet Draft draft-ietf-tsvwg-l4s-arch-03, Internet Engineering Task Force, Oct. 2018. (Work in Progress).

[10] CHENG, Y., AND CARDWELL, N. Making Linux TCP Fast. In *Proc. Netdev 1.2* (Oct. 2016).

[11] CHENG, Y., CARDWELL, N., DUKKIPATI, N., AND JHA, P. RACK: a time-based fast loss detection algorithm for TCP. Internet Draft draft-ietf-tcpm-rack-04, Internet Engineering Task Force, July 2018. (Work in Progress).

[12] CHENG, Y., CHU, J., RADHAKRISHNAN, S., AND JAIN, A. TCP Fast Open. Request for Comments 7413, RFC Editor, Dec. 2014.

[13] DE SCHEPPER, K., BRISCOE (ED.), B., ALBISSER, O., AND TSANG, I.-J. DualQ Coupled AQM for Low Latency, Low Loss and Scalable Throughput (L4S). Internet Draft draft-ietf-tsvwg-aqm-dualq-coupled-08, Internet Engineering Task Force, Nov. 2018. (Work in Progress).

[14] DE SCHEPPER, K., BRISCOE (ED.), B., AND TSANG, I.-J. Identifying Modified Explicit Congestion Notification (ECN) Semantics for Ultra-Low Queuing Delay (L4S). Internet Draft draft-ietf-tsvwg-ecn-l4s-id-05, Internet Engineering Task Force, Nov. 2018. (Work in Progress).

[15] HA, S., AND RHEE, I. Hybrid Slow Start for High-Bandwidth and Long-Distance Networks . In *Proc. Int'l Wkshp on Protocols for Future, Large-scale & Diverse Network Transports (PFLDNeT'08)* (2008).

[16] HOEILAND-JOERGENSEN, T., MCKENNEY, P., TÄHT, D., GETTYS, J., AND DUMAZET, E. The FlowQueue-CoDel Packet Scheduler and Active Queue Management Algorithm. Request for Comments RFC8290, RFC Editor, Jan. 2018.

[17] KÜHLEWIND, M. State of ECN and improving congestion feedback with AccECN in Linux. In *Proc. Netdev 2.2* (Dec. 2017).

[18] KÜHLEWIND, M., SCHEFFENEGGER, R., AND BRISCOE, B. Problem Statement and Requirements for Increased Accuracy in Explicit Congestion Notification (ECN) Feedback. Request for Comments RFC7560, RFC Editor, Aug. 2015.

[19] KÜHLEWIND, M., WALTER, M., LEARMOUTH, I., AND TRAMMELL, B. Tracing Internet Path Transparency. In *Proc. IFIP Network Traffic Measurement and Analysis Conference (TMA), 2018* (June 2018), IFIP.

[20] MISUND, J., AND BRISCOE, B. Paced Chirping - Rethinking TCP start-up. In *Proc. Netdev 0x13* (Mar. 2019).

[21] MISUND, J., AND BRISCOE, B. Paced Chirping: Rapid flow start with very low queuing delay. In *Proc. IEEE Global Internet Symp.* (May 2019), IEEE.

[22] RAMAKRISHNAN, K. K., FLOYD, S., AND BLACK, D. The Addition of Explicit Congestion Notification (ECN) to IP. Request for Comments 3168, RFC Editor, Sept. 2001.

[23] RHEE, I., XU, L., HA, S., ZIMMERMAN, A., EGGERT, L., AND SCHEFFENEGGER, R. CUBIC for Fast Long-Distance Networks. Request for Comments RFC8312, RFC Editor, Aug. 2018.