Workgroup:
Internet Congestion Control Research Group
(ICCRG)
Internet-Draft:
draft-briscoe-iccrg-prague-congestion-
control-01
Published: 11 July 2022
Intended Status: Experimental
Expires: 12 January 2023
Authors: K. De Schepper   O. Tilmans   B. Briscoe, Ed.
Nokia Bell Labs   Nokia Bell Labs   Independent

Prague Congestion Control

Abstract

This specification defines the Prague congestion control scheme, which is derived from DCTCP and adapted for Internet traffic by implementing the Prague L4S requirements. Over paths with L4S support at the bottleneck, it adapts the DCTCP mechanisms to achieve consistently low latency and full throughput. It is defined independently of any particular transport protocol or operating system, but notes are added that highlight issues specific to certain transports and OSs. It is mainly based on the current default options of the reference Linux implementation of TCP Prague, but it includes experience from other implementations where available. It separately describes non-default and optional parts, as well as future plans.

The implementation does not satisfy all the Prague requirements (yet) and the IETF might decide that certain requirements need to be relaxed as an outcome of the process of trying to satisfy them all. In two cases, research code is replaced by placeholders until full evaluation is complete.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

This document defines the Prague congestion control. It is defined independent of any particular transport protocol or operating system, but notes are added that highlight issues specific to certain transports and OSs. The authors are most familiar with the reference implementation of Prague on Linux over TCP. So that forms the basis of the large majority of platform-specific notes. Nonetheless, wherever possible, experience from implementers on other platforms is included, and the intention is to gather more into this document during the drafting process.

The Prague CC is intended to maintain consistently low queuing delay over network paths that offer L4S support at the bottleneck. Where the bottleneck does not support L4S, the Prague CC is intended to fall back to behaving like a conventional 'Classic' congestion control. L4S stands for Low Latency, Low Loss Scalable throughput. L4S support in the network involves Active Queue Management (AQM) with a very shallow target queueing delay (of the order of a millisecond) that applies immediate Explicit Congestion Notification (ECN). 'Immediate ECN' means that the network applies ECN marking based on the instantaneous queue, without any smoothing or filtering, The Prague CC takes on the job of smoothing and filtering the congestion signals from the network.

The Prague CC is a particular instance of a scalable congestion control, which is defined in Section 1.4. Scalable congestion control is the part of the L4S architecture that does the actual work of maintaining low queuing delay and ensuring that the delay and throughput properties scale with flow rate.

The L4S architecture [I-D.ietf-tsvwg-l4s-arch] places the host congestion control in the context of the other parts of the system. In particular the different types of L4S AQM in the network and the codepoints in the IP-ECN field that convey to the network that the host supports the L4S form of ECN. The architecture document also covers other issues such as: incremental deployment; protection of low latency queues against accidental or malicious disruption; and the relationship of L4S to other low latency technologies. The specification of the L4S ECN Protocol [I-D.ietf-tsvwg-ecn-l4s-id] sets down the requirements that the Prague CC has to follow (called the Prague L4S Requirements - see Section 2.1 for a summary).

Links to implementations of the Prague CC and other scalable congestion controls (all open source) can be found via the L4S landing page [L4S-home], which also links to numerous other L4S-related resources. A (slightly dated) paper on the specific implementation of the Prague CC in Linux over TCP is also available [PragueLinux], and the code is at [linux-code].

1.1. Motivation: Low Queuing Delay /and/ Full Throughput

The Prague CC is capable of keeping queuing delay consistently low while fully utilizing available capacity. In contrast, Classic congestion controls need to induce a reasonably large queue (approaching a bandwidth-delay product) in order to fully utilize capacity. Therefore, prior to scalable CCs like DCTCP and Prague, it was believed that very low delay was only possible by limiting throughput and isolating the low delay traffic from capacity-seeking traffic.

The Prague CC uses additive increase multiplicative decrease (AIMD), in which it increases its window until an ECN mark (or loss) is detected, then yields in a continual sawtooth pattern. The key to keeping queuing delay low without under-utilizing capacity is to keep the sawteeth tiny. For example the average duration of a Prague CC sawtooth is of the order of a round trip, whereas a classic congestion control sawtooths over hundreds of round trips. For instance, over an RTT of 36ms, at 100Mb/s CUBIC takes about 106 round trips to recover, and at 800 Mb/s its recovery time triples to over 340 round trips, or still more than 12 seconds (Reno would take 57 seconds.

Keeping the sawtooth amplitude down keeps queue variation down and utilization up. Keeping the duration of the sawteeth down ensures control remains tight. The definition of a scalable CC is that the duration between congestion marks does not increase as flow rate scales, all other factors being equal. This is important, because it means that the sawteeth will always stay tiny. So queue delay will remain very low, and control will remain very tight.

The tip of each sawtooth occurs when the bottleneck link emits a congestion signal. Therefore such small sawteeth are only feasible when ECN is used for the congestion signals. If loss were used, the loss level would be prohibitively high. This is why L4S-ECN has to depart from the requirement of Classic ECN[RFC3168] that an ECN mark is equivalent to a loss. Because otherwise the response to the high level of ECN marking would have to be as great as the response to an equivalent level of loss.

The Prague CC is derived from Data Center TCP (DCTCP [RFC8257]). DCTCP is confined to controlled environments like data centres precisely because it uses such small sawteeth, which induce such a high level of congestion marking. For a CC using Classic ECN, this would be interpreted as equivalent to the same, very high, loss level. The Classic CC would then continually drive its own rate down in the face of such an apparently high level of congestion.

This is why coexistence with existing traffic is important for the Prague CC. It has to be able to detect whether it is sharing the bottleneck with Classic traffic, and if so fall back to behaving in a Classic way. If the bottleneck does not support ECN at all, that is easy - the Prague CC just responds in the Classic way to loss (see Section 2.4.1). But if it is sharing the bottleneck with Classic ECN traffic, this is more difficult to detect (see Section 3.3). Because the Prague CC removes most of the queue, it also addresses RTT-dependence. Otherwise, at low base RTTs, its flow rate would become far more RTT-dependent than Classic CCs.

1.2. Document Purpose

There is not 'One True Prague CC'. L4S is intended to enable development of any scalable CC that meets the L4S Prague requirements [I-D.ietf-tsvwg-ecn-l4s-id]. This document attempts to describe a reference implementation and attempts to generalize it to different transports and OS platforms. The implementation does not satisfy all the Prague requirements (yet), and the IETF might decide that certain requirements need to be relaxed as an outcome of the process of trying to satisfy them all.

1.3. Maturity Status (To be Removed Before Publication)

The field of congestion control is always a work in progress. However, there are areas of the Prague CC that are still just placeholders while separate research code is evaluated. And in other implementations of the Prague CC, other areas are incomplete. In the Linux reference implementation of TCP Prague, interim code is used in the incomplete areas, which are:

* Flow start and restart (standard slow start is used, even though it often exits early in L4S environments were ECN marking tends to be frequent);

*Faster than additive increase (standard additive increase is used, which makes the flow particularly sluggish if it has dropped out of slow start early).

The body of this document describes the Prague CC as implemented. Any non-default options or any planned improvements are separated out into Section 3 on "Variants and Future Work". As each of the above areas is addressed, it will will be removed from this section and its description in the body of the document will be updated. Once all areas are complete, this section will be removed. Prague CC will then still be a work in progress, but only on a similar footing as all other congestion controls.

1.4. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119] when, and only when, they appear in all capitals, as shown here.

Definitions of terms:

Classic Congestion Control: A congestion control behaviour that can co-exist with standard TCP Reno [RFC5681] without causing significantly negative impact on its flow rate [RFC5033]. With Classic congestion controls, as flow rate scales, the number of round trips between congestion signals (losses or ECN marks) rises with the flow rate. So it takes longer and longer to recover after each congestion event. Therefore control of queuing and utilization becomes very slack, and the slightest disturbance prevents a high rate from being attained [RFC3649].

Scalable Congestion Control: A congestion control where the average time from one congestion signal to the next (the recovery time) remains invariant as the flow rate scales, all other factors being equal. This maintains the same degree of control over queueing and utilization whatever the flow rate, as well as ensuring that high throughput is robust to disturbances. For instance, DCTCP averages 2 congestion signals per round-trip whatever the flow rate. For the public Internet a Scalable transport has to comply with the requirements in Section 4 of [I-D.ietf-tsvwg-ecn-l4s-id] (aka. the 'Prague L4S requirements').

Response function: The relationship between the window (cwnd) of a congestion control and the congestion signalling probability, p, in steady state. A general response function has the form cwnd = K/p^B, where K and B are constants. In an approximation of the response function of the standard Reno CC, B=1/2. For a scalable congestion control B=1, so its response function takes the form cwnd = K/p. The number of congestion signals per round is p*cwnd, which equates to the constant, K, for a scalable CC. Hence the definition of a scalable CC above.

Reno-friendly: The subset of Classic traffic that excludes unresponsive traffic and excludes experimental congestion controls intended to coexist with Reno but without always being strictly friendly to it (as allowed by [RFC5033]). Reno-friendly is used in place of 'TCP-friendly', given that the TCP protocol is used with many different congestion control behaviours.

Classic ECN: The original Explicit Congestion Notification (ECN) protocol [RFC3168], which requires ECN signals to be treated the same as drops, both when generated in the network and when responded to by the sender.

The names used for the four codepoints of the 2-bit IP-ECN field are as defined in [RFC3168]: Not ECT, ECT(0), ECT(1) and CE, where ECT stands for ECN-Capable Transport and CE stands for Congestion Experienced.

A packet marked with the CE codepoint is termed 'ECN-marked' or sometimes just 'marked' where the context makes ECN obvious.

CC: Congestion Control

ACK: an ACKnowledgement, or to ACKnowledge

EWMA: Exponentially Weighted Moving Average

RTT: Round Trip Time

Definitions of Parameters and Variables:

MTU_BITS: Maximum transmission unit [b]

cwnd: Congestion window [B]

ssthresh: Slow start threshold [B]

inflight: The amount of data that the sender has sent but not yet received ACKs for [B]

p: Steady-state probability of drop or marking []

alpha: EWMA of the ECN marking fraction []

acked_sacked: the amount of new data acknowledged by an ACK [B]

ece_delta: the amount of newly acknowledged data that was ECN-marked [B]

ai_per_rtt: additive increase to apply per RTT [B]

srtt: Smoothed round trip time [s]

MAX_BURST_DELAY: Maximum allowed bottleneck queuing delay due to segmentation offload bursts [s] (default 0.25 ms for the public Internet)

2. Prague Congestion Control

2.1. The Prague L4S Requirements

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 day after the first public demonstration of L4S.

The meeting agreed a list of modifications to DCTCP [RFC8257] 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. This list was adopted by the IETF, and has continued to evolve (see section 4 of [I-D.ietf-tsvwg-ecn-l4s-id]). The requirements are no longer TCP-specific, applying irrespective of wire-protocol (TCP, QUIC, RTP, SCTP, etc).

This unusual start to the life of the project led to the unusual development process of a reference implementation that had to resolve a number of ambitious requirements, already known to be in tension [Tensions17].

DCTCP already implements a scalable congestion control. So most of the changes to make it usable over the Internet seemed trivial, some 'merely' involving adoption of other parallel developments like Accurate ECN TCP feedback [I-D.ietf-tcpm-accurate-ecn] or RACK [RFC8985]. Others have been more challenging (e.g. RTT-independence). And others that seemed trivial became challenging given the complex set of bugs and behaviours that characterize today's Internet and the Linux stack.

The more critical implementation challenges are highlighted in the following sections, in the hope we can prevent mistakes being repeated (see for instance Section 2.3.2, Section 2.4.2). There was also a set of five intertwined 'bugs' - all masking each other, but causing unpredictable or poor performance as different code modifications unmasked them. A draft write-up about these has been prepared, which is longer than the whole of the present document, so it will be included by reference once published.

During the development process, we have unearthed fundamental aspects of the implementation and indeed the design of DCTCP and Prague that have still not caught up with the paradigm shift from existence to extent-based congestion response. Some have been implemented by default, e.g. not suppressing additive increase for a round trip after a congestion event (Section 2.4.3). Others have been implemented but not fully evaluated, e.g. removing the 1-2 unnecessary round trips of lag in feedback processing (Section 3.1.3) and yet others are still future plans, e.g. further RTT-independence (Section 3.4) and exploiting combined congestion metrics in more cases (Section 3.2).

The requirements are categorized into those that would impact other flows if not handled properly and performance optimizations that are important but optional from the IETF's point of view, because they only affect the flow itself. The list below maps the order of the requirements in [I-D.ietf-tsvwg-ecn-l4s-id] to the order in this document (which is by functional categories and code status):

Mandatory or Advisory Requirements:

* L4S-ECN packet identification: use of ECT(1) (Section 2.2)

* Accurate ECN feedback (Section 2.3.1)

* Reno-friendly response to a loss (Section 2.4.1)

* Detection of a classic ECN AQM (Section 3.3)

* Reduced RTT dependence (Section 2.4.4)

* Scaling down to a fractional window (no longer mandatory, see Section 3.5)

*Detecting loss in units of time (Section 2.3.3)

*Minimizing bursts (Section 2.5.1

Optional*performance optimizations: ECN-capable control packets (Section 2.2)

*Faster flow start (Section 3.1.1)

*Faster than additive increase (Section 3.1.2)

*Segmentation offload (Section 2.5.2)

2.2. Packet Identification

On the public Internet, a sender using the Prague CC MUST set the ECT(1) codepoint on all the packets it sends, in order to identify itself as an L4S-capable congestion control (Req 4.1 [I-D.ietf-tsvwg-ecn-l4s-id]).

This applies whatever the transport protocol, whether TCP, QUIC, RTP, etc. In the case of TCP, unlike an RFC 3168 TCP ECN transport, a sender can set all packets as ECN-capable, including TCP control packets and retransmissions [RFC8311], [I-D.ietf-tcpm-generalized-ecn].

The Prague CC SHOULD optionally be configurable to use the ECT(0) codepoint in private networks, such as data centres, which might be necessary for backward compatibility with DCTCP deployments where ECT(1) might already have another usage.

Implementation note:

TCP Prague in Linux kernel: The kernel was updated to allow the ECT(1) flag to be set from within a CC module. The Prague CC then has full control over the ECN code point it uses at any one time. In this way it enforces the use of ECT(1) (or optionally ECT(0)) and non-ECT when required.

2.3. Detecting and Measuring Congestion

2.3.1. Accurate ECN Feedback

When feedback of ECN markings was added to TCP [RFC3168], it was decided not to report any more than one mark per RTT. L4S-capable congestion controls need to know the extent, not just the existence of congestion (Req 4.2. [I-D.ietf-tsvwg-ecn-l4s-id]). Recently defined transports (DCCP, QUIC, etc) typically already satisfy this requirement. So they are dealt with separately below, while TCP and derivatives such as SCTP [RFC4960] are covered first.

2.3.1.1. Accurate ECN Feedback with TCP & Derivatives

The TCP wire protocol is being updated to allow more accurate feedback (AccECN [I-D.ietf-tcpm-accurate-ecn]). Therefore, in the case where a sender uses the Prague CC over TCP, whether as client or server:

* it MUST itself support AccECN;
* to support AccECN it also has to check that its peer supports AccECN during the handshake.

If the peer does not support accurate ECN feedback, the sender MUST fall back to a Reno-friendly CC behaviour for the rest of the connection. The non-Prague TCP sender MUST then no longer set ECT(1) on the packets it sends. Note that the peer only needs to support AccECN; there is no need (and no way) to find out whether the peer is using an L4S-capable congestion control.

Note that a sending TCP client that uses the Prague CC can set ECT(1) on the SYN prior to checking whether the other peer supports AccECN (as long as it follows the procedure in [I-D.ietf-tcpm-generalized-ecn] if it discovers the peer does not support AccECN).

Implementation note:

TCP Prague in Linux kernel: The kernel had been updated to support AccECN Independent of the CC module in use. So the kernel tries to negotiate AccECN exchange whichever congestion control module is selected. An additional check is provided to verify that the kernel actually does support AccECN, based on which the Prague CC module will decide to proceed using scalable CC or fall back to a Classic CC (Reno in the current implementation).

A system wide option is available to disable AccECN negotiation, but the Prague CC module will always override this setting, as it depends on AccECN. Then, solely in this case, AccECN will only be active for TCP flows using the Prague CC.

2.3.1.2. Accurate ECN Feedback with Other Modern Transports

Transport protocols specified recently, .e.g. DCCP [RFC4340], QUIC [RFC9000], are unambiguously suitable for Prague CCs, because they were designed from the start with accurate ECN feedback.

In the case of RTP/RTCP, ECN feedback was added in [RFC6679], which is sufficient for the Prague CC. However, it is preferable to use the most recent improvements to ECN feedback in [RFC8888], as used in the implementation of the L4S variant of SCReAM [RFC8298].

2.3.2. Moving Average of ECN Feedback

The Prague CC currently maintains a moving average of ECN feedback in a similar way to DCTCP. This section is provided mainly because performance has proved to be sensitive to implementation precision in this area. So first, some background is necessary.

The Prague CC triggers update of its moving average once per RTT by recording the packet it sent after the previous update, then watching for the ACK of that packet to return. To maintain its moving average, it measures the fraction, frac, of ACKed bytes that carried ECN feedback over the previous round trip. It then updates an exponentially weighted moving average (EWMA) of this fraction, called alpha, using the following algorithm:

```
alpha += g * (frac - alpha);
```

where g is the gain of the EWMA (default 1/16).

The moving average, alpha, is initialized to 1 at the first sign of ECN feedback, which ensures the maximum congestion response to the first appearance of congestion at a bottleneck supporting ECN.

Implementation notes:

Rounding problems in DCTCP: Alpha is a fraction between 0 and 1, and it needs to be represented with high resolution because the larger the bandwidth-delay product (BDP) of a flow, the smaller the value that alpha converges to (in steady state alpha = 2/cwnd). In principle, Linux DCTCP maintains the moving average 'alpha' using the same formula as Prague CC uses (as above). Linux represents alpha with a 10-bit integer (with resolution 1/1024). However, up to kernel release 3.19, Linux used integer arithmetic that could not reduce alpha below 15/1024. Then it was patched so that any value below 16/1024 was rounded down to zero [patch-alpha-zero]. For a flow with a higher BDP than 128 segments, this means that, alpha flip-flops. Once it has flopped down to zero DCTCP becomes unresponsive until it has built sufficient queue to flip up to 16/1024. For larger BDPs, this causes DCTCP to induce larger sawteeth, which loses the low-queuing-delay and high-utilization intent of the algorithm.

Upscaled alpha in Prague CC: To resolve the above problem the implementation of TCP Prague in Linux maintains upscaled_alpha = alpha/g instead of alpha:

```
upscaled_alpha += frac - g * upscaled_alpha;
```

This technique is the same as Linux uses for the retransmission timer variables, srtt and mdev. Prague CC also uses 20 bits for alpha,

Currently the above per-RTT update to the moving average, which was inherited from DCTCP, is the default in the Prague CC. However, another approach is being investigated because these per-RTT updates introduce 1--2 rounds of delay into the congestion response on top of the inherent round of feedback delay (see Section 3.1.3 in the section on variants and future work).

2.3.3. Scaling Loss Detection with Flow Rate

After an ACK leaves a gap in the sequence space, a Prague CC is meant to deem that a loss has occurred using 'time-based units' (Req 4.3. [I-D.ietf-tsvwg-ecn-l4s-id]). This is in contrast to the traditional approach that counts a hard-coded number of duplicate ACKs, e.g. the 3 Dup-ACKs specified in [RFC5681]. Counting packets rather than time unnecessarily tightens the time within which parallelized links have to keep packets in sequence as flow rate scales over the years.

To satsify this requirement, a Prague CC SHOULD wait until a certain fraction of the RTT has elapsed before it deems that the gap is due to packet loss. The reference implementation of TCP Prague in Linux uses RACK [RFC8985] to address this requirement. An approach similar to TCP RACK is also used in QUIC.

At the start of a connection, RACK counts 3 DupACKs to detect loss because the initial smoothed RTT estimate can be inaccurate. This would depend indirectly on time as long as the initial window (IW) is paced over a round trip (see Section 2.4.5). For instance, if the initial window of 10 segments was paced evenly across the initial RTT then, in the next round, an implementation that deems there has been a loss after (say) 1/4 of an RTT can count 1/4 of 10 = 3 DupACKs (rounded up). Subsequently, as the window grows, RACK shifts to using a fraction of the RTT for loss detection.

2.4. Congestion Response Algorithm

In congestion avoidance phase, a Prague CC uses a similar additive increase multiplicative decrease (AIMD) algorithm to DCTCP, but with the following differences:

2.4.1. Loss behaviour

A Prague CC MUST use a Reno-friendly congestion response (such as that of CUBIC [I-D.ietf-tcpm-rfc8312bis] or Reno [RFC5681]) on detection of a loss (Req 2 in section 4.3. of [I-D.ietf-tsvwg-ecn-l4s-id]). DCTCP falls back to Reno for the round trip after a loss, and the Linux reference implementation of TCP Prague currently inherits this behaviour. On detection of loss, an implementation can use CUBIC's behaviour instead of Reno's for both the reduction after the loss and the subsequent growth of cwnd until the next congestion event.

If a Prague CC has already reduced the congestion window due to ECN feedback less than a round trip before it detects a loss, it MAY reduce the congestion window by a smaller amount due to the loss, as long as the reductions, due to ECN and the loss, when multiplied together result in the reduction that the implementation usually makes in response to loss (e.g. 50% to emulate Reno or 30% to emulate CUBIC).

See Section 3.2 for discussion of future work on congestion control using a combination of delay, ECN and loss.

Implementation note:

DCTCP bug prior to v5.1: A Prague CC cannot rely on inheriting the fall-back-on-loss behaviour of the DCTCP code in the Linux kernel prior to v5.1, due to a previous bug in the fast retransmit code (but not in the retransmission timeout code) [patch-loss-react].

2.4.2. Multiplicative Decrease on ECN Feedback

The Prague CC currently responds to ECN feedback in a similar way to DCTCP. This section is provided mainly because performance has proved to be sensitive to implementation details in this area. So the following recap of the congestion response is needed first.

As explained in Section 2.3.2, the Prague CC (like DCTCP) clocks its moving average of ECN-marking, alpha, once per round trip throughout a connection. Nonetheless, it only triggers a multiplicative decrease to its congestion window when it actually receives an ACK carrying ECN feedback. Then it suppresses any further decreases for one round trip, even if it receives further ECN feedback. This is termed Congestion Window Reduced or CWR state.

The Prague CC (like DCTCP) ensures that the average recovery time remains invariant as flow rate scales (Req 4.3 of [I-D.ietf-tsvwg-ecn-l4s-id]) by making the multiplicative decrease depend on the prevailing value of alpha as follows:

```
ssthresh = (1 - alpha/2) * cwnd;
```

Implementation notes:

Upscaled alpha: With reference to the earlier discussion of integer arithmetic precision (Section 2.3.2), alpha = g * upscaled_alpha.

Carry of fractional cwnd remainder: Typically the absolute reduction in the window is only a small number of segments. So, if the Prague CC implementation counts the window in integer segments (as in the Linux reference code), delay can be made significantly less jumpy by tracking a fractional value alongside the integer window and carrying over any fractional remainder to the next reduction. Also, integer rounding bias ought to be removed from the multiplicative decrease calculation.

In dynamic scenarios, as flows find a new operating point, alpha will have often tailed away to near-nothing before the onset of congestion. Then DCTCP's tiny reduction followed by no further response for a round is precisely the wrong way for a CC to respond. A solution to this problem is being evaluated as part of the work already mentioned to improve Prague's responsiveness (see Section 3.1.3 in the section on variants and future work).

2.4.3. Additive Increase and ECN Feedback

Unlike DCTCP, the Prague CC does not suppress additive increase for one round trip after a congestion window reduction (while in CWR state). Instead, a Prague CC applies additive increase irrespective of its CWR state, but only for bytes that have been ACK'd without ECN feedback. Specifically, on each ACK,

```
 cwnd += (acked_sacked - ece_delta) * ai_per_rtt / cwnd;
```

where:

acked_sacked is the number of new bytes acknowledged by the ACK;

ece_delta is the number of newly acknowledge ECN-marked bytes;

ai_per_rtt is a scaling factor that is typically 1 SMSS except for small RTTs (see Section 2.4.4)

Superficially, the traditional suppression of additive increase for the round after a decrease seems to make sense. However, DCTCP and Prague are designed to induce an average of 2 congestion marks per RTT in steady state, which leaves very little space for any increase between the end of one round of CWR and the next mark. In tests, when a test version of Prague CC is configured to completely suppress additive increase during CWR (like Reno and DCTCP), it sawteeth become more irregular, which is its way of making some decreases large enough to open up enough space for an increase. This irregularity tends to reduce link utilization. Therefore, the reference Prague CC continues additive increase irrespective of CWR state.

Nonetheless, rather than continue additive increase regardless of congestion, it is safer to only increase on those ACKs that do not feed back congestion. This approach reduces additive increase as the marking probability increases, which tends to keep the marking level unsaturated (below 100%) (see Section 3.1 of [Tensions17]). Under stable conditions, Prague's congestion window then becomes proportional to (1-p)/p, rather than 1/p.

See also 'Faster than Additive Increase' (Section 3.1.2)

2.4.4. Reduced RTT-Dependence

The window-based AIMD described so far was inherited from Reno via DCTCP. When many long-running Reno flows share a link, their relative packet rates become roughly inversely proportional to RTT (packet rate =~ 1/RTT). Then a flow with very small RTT will dominate any flows with larger RTTs.

Queuing delay sets a lower limit to the smallest possible RTT. So, prior to the extremely low queuing delay of L4S, extreme cases of RTT dependence had never been apparent. Now that L4S has removed most of the queuing delay, we have to address the root-cause of RTT-dependence, which the Prague CC is required to do, at least when the RTT is small (see the 'Reduced RTT bias' aspect of Req 4.3. [I-D.ietf-tsvwg-ecn-l4s-id]). Here, a small RTT is defined as below the typical RTT for the intended deployment environment.

The reference Prague CC reduces RTT bias by using a virtual RTT (rtt_virt) rather than the actual smoothed RTT (srtt) for all three of: i) the period of additive window increase; ii) the EWMA update period; and iii) the duration of CWR state after a decrease. rtt_virt is calculated as a function of the actual smoothed RTT, chosen so that, when the srtt is high, the virtual RTT is essentially the same; but for lower actual RTTs, the virtual RTT is increasingly larger than the actual RTT. Example functions for the virtual RTT are:

```
   rtt_virt = max(srtt, RTT_VIRT_MIN);

   rtt_virt = srtt + AdditionalRTT;
```

where RTT_VIRT_MIN and AdditionalRTT are constants. The current default is rtt_virt = max(srtt, 25ms), which addresses the main Prague requirement for when the RTT is smaller than typical.

As the actual window (cwnd) is still sent within 1 actual RTT, we also need to use a (conceptual) virtual window, cwnd_virt. For instance, if rtt_virt = 25 ms then, when the actual RTT is 5 ms, there are rtt_virt/srtt = 5 times more packets in cwnd_virt, than in the actual window, cwnd, because it spans 5 actual round trips. We define M as the ratio rtt_virt/srtt.

In Reno or DCTCP, additive increase is implemented by dividing the desired increase of 1 segment per round over the cwnd packets in the round. This requires an increase of 1/cwnd per packet. In the Linux implementation of TCP Prague, the aim is to increase the reference window by 1 segment over a virtual RTT. However, in practice the increase is applied to the actual window, cwnd, which is M times smaller than cwnd_virt. So cwnd has to be increased by only 1/M segments over rtt_virt. But again, in practice, the increase is applied over an actual window of packets spanning an actual RTT, which is also M times smaller than the virtual RTT. So the desired increase in cwnd is only 1/M^2 segments over an actual round trip containing cwnd packets. Therefore, the increase in cwnd per packet has to be (1/M^2) * (1/cwnd).

Unless a flow lasts long enough for rates to converge, aiming for equal rates will not be relevant. So, the Reduced RTT-Dependence algorithm only comes into effect after D rounds, where D is configurable (current default 500). Continuing the previous example, if actual srtt=5 ms and rtt_virt = 25 ms, then Prague would use the regular RTT-dependent algorithm for the first 500*5ms = 2.5s. Then it would start to converge to more equal rates using its Reduced RTT-Dependence algorithm. If the actual RTT were higher (e.g. 20ms), it would stay in the regular RTT-dependent mode for longer (500 rounds = 10s), but this would be mitigated by the actual RTT it uses at the start being closer to the virtual RTT it eventually uses (20ms and 25ms resp.).

This approach prevents reduced RTT-dependence from making the flow less responsive at start-up and ensures that its early throughput share is based on its actual RTT. The benefit is that short flows (mice) give themselves priority over longer flows (elephants), and shorter RTTs will still converge faster than longer RTTs. Nonetheless, the throughput still converges to equal rates after D rounds.

It is planned to reset the algorithm to the regular RTT-dependent behaviour after an idle, not just at flow start, as discussed under Future Work in Section 3.4.

Section 3.4 also discusses extending the reduction in RTT-dependence to longer RTTs than RTT_VIRT_MIN (i.e. longer than 25ms). The current Prague implementation does not support this.

2.4.5. Flow Start or Restart

Currently the Linux reference implementation of TCP Prague uses the standard Linux slow start code. Slow start is exited once a single mark is detected.

When other flows are actively filling the link, regular marks are expected, causing slow start of new flows to end prematurely. This is clearly not ideal, so other approaches are being worked on (see Section 3.1.1). However, slow start has been left as the default until a properly matured solution is completed.

2.5. Packet Sending

2.5.1. Packet Pacing

The Prague CC SHOULD pace the packets it sends to avoid the queuing delay and under-utilization that would otherwise be caused by bursts of packets that can occur, for example, when a jump in the acknowledgement number opens up cwnd. Prague does this in a similar way to the base Linux TCP stack, by spacing out the window of packets evenly over the round trip time, using the following calculation of the pacing rate [b/s]:

```
pacing_rate = MTU_BITS * max(cwnd, inflight) / srtt;
```

During slow start, as in the base Linux TCP stack, Prague factors up pacing_rate by 2, so that it paces out packets twice as fast as they are acknowledged. This keeps up with the doubling of cwnd, but still prevents bursts in response to any larger transient jumps in cwnd.

```
 if (cwnd < ssthresh / 2)
     pacing_rate *= 2;
```

During congestion avoidance, the Linux TCP Prague implementation does not factor up pacing_rate at all. This contrasts with the base Linux TCP stack, which currently factors up pacing_rate by a ratio parameter set to 1.2. The developers of the base Linux stack confirmed that this factor of 1.2 was only introduced in case it improved performance, but there were no scenarios where it was known to be needed. In testing of Prague, this factor was found to cause queue delay spikes whenever cwnd jumped more than usual. And throughput was no worse without it. So it was removed from the TCP Prague CC.

The Prague CC can use alternatives to the traditional slow-start algorithm, which use different pacing (see Section 2.4.5).

2.5.2. Segmentation Offload

In the absence of hardware pacing, it becomes increasingly difficult for a machine to scale to higher flow rates unless it is allowed to send packets in larger bursts, for instance using segmentation offload. Happily, as flow rate scales up, proportionately more packets can be allowed in a burst for the same amount of queuing delay at the bottleneck.

Therefore, the Prague CC sends packets in a burst as long as it will not induce more than MAX_BURST_DELAY of queuing at the bottleneck. From this constant and the current pacing_rate, it calculates how many MTU-sized packets to allow in a burst:

```
max_burst = pacing_rate * MAX_BURST_DELAY / MTU_BITS
```

The current default in the Linux TCP Prague for MAX_BURST_DELAY is 250us which supports marking thresholds starting from about 500us without underutilization. This approach is similar to that in the Linux TCP stack, except there MAX_BURST_DELAY is 1ms.

3. Variants and Future Work

3.1. Getting up to Speed Faster

Appendix A.2. of [I-D.ietf-tsvwg-ecn-l4s-id] outlines the performance optimizations needed when transplanting DCTCP from a DC environment to a wide area network. The following subsections address two of those points: faster flow startup and faster than additive increase. Then Section 3.1.3 covers the flip side, in which established flows have to yield faster to make room, otherwise queuing will result.

3.1.1. Flow Start (or Restart)

The Prague performance For faster flow start, two approaches are currently being investigated in parallel:

Modified Slow Start: The traditional exponential slow start can be modified both at the start and the end, with the aim of reducing the risk of queuing due to bursts and overshoot:

Pacing IW: A Prague CC can use an initial window of 10 (IW10 [RFC6928]), but pacing of this Initial Window is recommended to try to avoid the pulse of queuing that could otherwise occur. Pacing IW10 also spreads the ACKs over the round trip so that subsequent rounds consist of ten subsets of packets (with 2, 4, 8 etc. per round in each subset), rather than a single set with 20, 40, 80 etc. in each round. With IW paced, if a queue builds during a round (e.g. due to other unexpected traffic arriving) it can drain in the gap before the next subset, rather than the whole set backing up into a much larger queue. As smoothed RTT is unknown or inaccurate at the start of a flow, an implementation can pace IW over a fraction of the initial smoothed RTT (perhaps also clamped between hard-coded sanity limits). The implementation could also initialize SRTT with a value it had previously cached per destination (as long as it is sufficiently fresh). The safety factor could depend on whether a cached value was available and how recent it was.

In the Linux reference implementation of TCP Prague, IW pacing can be optionally enabled, but it is off by default, because it is yet to be fully evaluated. It currently paces IW over half the initial smoothed round trip time (SRTT) measured during the handshake. SRTT is halved because the RTT often reduces after the initial handshake. For example: i) some CDNs move the flow to a closer server after establishment; ii) the initial RTT from a server can include the time to wake a sleeping handset battery; iii) some uplink technologies take a link-level round trip to request a scheduling slot.

It is also planned to exploit any cached knowledge of the path RTT to improve the initial estimate, for instance using the Linux per-destination cache. it is also planned to allow the application to give an RTT hint (by setting sk_max_pacing_rate in Linux) if the developer has reason to believe that the application has a better estimate.

Exiting slow start more gracefully: In the wide area Internet (in contrast to data centres), bottleneck access links tend to have much less capacity than the line rate of the sender. With a shallow immediate ECN threshold at this bottleneck, the slightest burst can tend to induce an ECN mark, which traditionally causes slow start to exit. A more gradual exit is being investigated for a Prague CC using the extent of marking, not just the existence of a single mark. This will be more consistent with the extent-based marking that scalable congestion controls use during congestion avoidance. Delay measurements (similar to Hystart++ [I-D.ietf-tcpm-hystartplusplus]) can also be used to complement the ECN signals.

Paced Chirping: In this approach, the aim is to both increase more rapidly than exponential slow-start and to greatly reduce any overshoot. It is primarily a delay-based approach, but the aim is also to exploit ECN signals when present (while not forgetting loss either). Therefore Paced Chirping is generally usable for any congestion control - not solely for Prague CC and L4S.

Instead of only aiming to detect capacity overshoot at the end of flow-start, brief trains of rapidly decreasing inter-packet spacing called chirps are used to test many rates with as few packets and as little load as possible. A full description is beyond the scope of this document. [LinuxPacedChirping] introduces the concepts and the code as well as citing the main papers on Paced Chirping.

Paced chirping works well over continuous links such as Ethernet and DSL. But better averaging and noise filtering are necessary over discontinuous link technologies such as WiFi, LTE cellular radio, passive optical networks (PON) and data over cable (DOCSIS). This is the current focus of this work.

The current Linux implementation of TCP Prague does not include Paced Chirping, but research code is available separately in Linux and ns3. it is accessible via the L4S landing page [L4S-home].

3.1.2. Faster than Additive Increase

The Prague CC has a startup phase and congestion avoidance phase like traditional CCs. In steady-state during congestion avoidance, like all scalable congestion controls, it induces frequent ECN marks, with the same average recovery time between ECN marks, no matter how much the flow rate scales.

If available capacity suddenly increases, e.g. other flow(s) depart or the link capacity increases, these regular ECN marks will stop. Therefore after a few rounds of silence (no ECN marks) in congestion avoidance phase, the Prague CC can assume that available capacity has increased, and switch to using the techniques from its startup phase (Section 3.1.1) to rapidly find the new, faster operating point. Then it can shift back into its congestion avoidance behaviour.

That is the theory. But, as explained in Section 3.1.1, the startup techniques, specifically paced chirping, are still being developed for discontinuous link types. Once the startup behaviour is available, the Linux implementation of the Prague CC will also have a faster than additive increase behaviour. S.3.2.3 of [PragueLinux]) gives a brief preview of the performance of this approach over an Ethernet link type in ns3.

3.1.3. Remove Lag in Congestion Response

To keep queuing delay low, new flows can only push in fast if established flows yield fast. It has recently been realized that the design of the Prague EWMA and congestion response introduces 1-2 rounds of lag (on top of the inherent round of feedback delay due to the speed of light). These lags were inherited from the design of DCTCP (see Section 2.3.2 and Section 2.4.2), where a couple of extra hundred microseconds was less noticeable. But congestion control in the wide area Internet cannot afford up to 2 rounds trips of extra lag.

To be clear, lag means delay before any response at all starts. That is qualititatively different from the smoothing gain of an EWMA, which /reduces/ the response by the gain factor (1/16 by default) in case a change in congestion does not persist. Smoothing gain can always be increased. But 1-2 rounds of lag means that, when a new flow tries to push in, the sender of an established flow will not respond /at all/ for 1-2 rounds after it first receives congestion feedback.

The Prague CC spends the first round trip of this lag gathering feedback to measure frac before it is input into the EWMA algorithm (see Section 2.3.2). Then there is up to one further round of delay because the implementations of DCTCP and Prague did not fully adopt the paradigm shift to extent-based marking - the timing of the decrease is still based on Reno.

Both Reno and DCTCP/Prague respond immediately on the first sign of congestion. Reno's response is large, so it waits a round in CWR state to allow the response to take effect. DCTCP's response is tiny (extent-based), but then it still waits a round in CWR state. So it does next-to-nothing for a round.

New EWMA and resposne algorithms to remove these 1-2 extra rounds of lag are described in [PerAckEWMA]. They have been implemented in Linux and an iterative process of evaluation and redesign is in progress. The EWMA is updated per-ACK, but it still changes as if it is clocked per round trip. The congestion response is still triggered by the first indication of ECN feedback, but it proceeds over the subsequent round trip so that it can take into account further incoming feedback as the EWMA evolves. The reduction is applied per-ACK but sized to result as if it had been a single response per round trip.

3.2. Combining Congestion Metrics

Ultimately, it would be preferable to take an integrated approach and use a combination of ECN, loss and delay metrics to drive congestion control. For instance, using a downward trend in ECN marking and/or delay as a heuristic to temper the response to loss. Such ideas are not in the immediate plans for the Linux TCP Prague, but some more specific ideas are highlighted in the following subsections.

3.2.1. ECN with Loss

If the bottleneck is ECN-capable, a loss due to congestion is very likely to have been preceded by a period of ECN marking. When the current Linux TCP Prague CC detects a loss, like DCTCP, it halves cwnd, even if it has already reduced cwnd in the same round trip due to ECN marking. This double reduction can end up factoring down cwnd to as little as 1/4 in one round trip.

On a loss while in CWR state following an ECN reduction, for an implementation that uses Reno response, it would be possible to use a decrease factor of 1/(2-alpha), which would compound with the previous decrease factor of (1-alpha/2) to result in a factor of: (1 - alpha/2) / (2-alpha)) = 1/2. In integer arithmetic, this division would be possible but relatively expensive. A less expensive alternative would be a decrease factor of (2+alpha)/4, which approximates to a compounded decrease factor of 1/2 for typical low values of alpha, even up to 30%. The compound decrease factor is never greater than 1/2 and in the worst case, if alpha were 100%, it is 3/8.

If an implementation uses a CUBIC response on loss after an ECN reduction in the same round trip, it can use a multiplicative decrease factor of 7/(5*(2-alpha)) which would result in a combined reduction to 7/10 of the previous cwnd, as intended for CUBIC.

3.2.2. ECN with Delay

Section 3.1.2 described the plans to shift between using ECN when close to the operating point and using delay by injecting paced chirps to find a new operating after the ECN signal goes silent for a few rounds. Paced chirping shifts more slowly to the new operating point the more noise there is in the delay measurements. Work is ongoing on treating any ECN marking as a complementary metric. The resulting less noisy combined metric should then allow the controller to shift more rapidly to each new operating point.

An alternative would be to combine ECN with the BBR approach, which induces a much less noisy delay signal by using less frequent but more pronounced delay spikes. The approach currently being taken is to adapt the chirp length to the degree of noise, so the chirps only become longer and/or more pronounced when necessary, for instance when faced with a discontinuous link technology such as WiFi. With multiple chirps per round, the noise can still be filtered out by averaging over them all, rather than trying to remove noise from each spike. This keeps the 'self-harm' to the minimum necessary, and ensures that capacity is always being sampled, which removes the risk of going stale.

3.3. Fall-Back on Classic ECN

The implementation of TCP Prague CC in Linux includes an algorithm to detect a Classic ECN AQM and fall back to Reno as a result, as required by the 'Coexistence with Classic ECN' aspect of the Prague Req 4.3. [I-D.ietf-tsvwg-ecn-l4s-id].

The algorithm currently used (v2) is relatively simple, but rather than describe it here, full rationale, pseudocode and explanation can be found in the technical report about it [ecn-fallback]. This also includes a selection of the evaluation results and a link to visualizations of the full results online. The current algorithm nearly always detects a Classic ECN AQM, and in the majority of the wide range of scenarios tested it is good at detecting an L4S AQM. However, it wrongly identifies and L4S AQM as Classic in a significant minority of cases when the link rate is low, or the RTT is high. The report gives ideas on how to improve detection in these scenarios, but in the mean time the algorithm has been disabled by default.

Recently, the report has been updated to include new ideas on other ways to distinguish Classic from L4S AQMs. The interested reader can access it themselves, so this living document will not be further summarized here.

3.4. Further Reduced RTT-Dependence

The algorithm to reduce RTT dependence is only relevant for long-running flows. So in the current TCP Prague implementation it remains disabled for a certain number of round trips after the start of a flow, as explained in Section 2.4.4. Instead, it would be possible to make rtt_virt gradually move from the actual RTT to the target virtual RTT, or perhaps depend on other parameters of the flow. Nonetheless, just switching in the algorithm after a number of rounds works well enough. It is planned to also disable the algorithm for a similar duration if a flow becomes idle then restarts, but this is yet to be evaluated.

Prague Req 4.3. in [I-D.ietf-tsvwg-ecn-l4s-id]) only requires reduced RTT bias "in the range between the minimum likely RTT and typical RTTs expected in the intended deployment scenario". Nonetheless, in future it would be preferable to be able to reduce the RTT bias for high RTT flows as well.

If a step AQM is used, the congestion episodes of flows with different RTTs tend to synchronize, which exacerbates RTT bias. To prevent this two candidate approaches will need to be investigated: i) It might be sufficient to deprecate step AQMs for L4S (they are not the preferred recommendation in [I-D.ietf-tsvwg-aqm-dualq-coupled]); or ii) the virtual RTT approach of Section 2.4.4 might be usable for higher than typical RTTs as well as lower. In this latter case, (srtt/rtt_virt)^2 segments would need to be added to the window per actual RTT. The current TCP Prague implementation does not support this faster AI for RTTs longer than RTT_VIRT_MIN (25ms), due to the expected (but unverified) impact on latency overshoot and responsiveness.

3.5. Scaling Down to Fractional Windows

A modification to v5.0 of the Linux TCP stack that scales down to sub-packet windows is available for research purposes via the L4S landing page [L4S-home]. The L4S Prague Requirements in section 4.3 of [I-D.ietf-tsvwg-ecn-l4s-id] recommend but no longer mandate scaling down to sub-packet windows. This is because becoming unresponsive at a minimum window is a tradeoff between protecting against other unresponsive flows and the extra queue you induce by becoming unresponsive yourself. So this code is not maintained as part of the Linux implementation of TCP Prague.

Firstly, the stack ahs to be modifed to maintain a fractional congestion window. The because the ACK clock cannot work below 1 packet per RTT, the code sets the time to send each packet, then readjusts the timing as each ACK arrives (otherwise any queuing accumulates a burst in subsequent rounds). Also, additive increase of one segment does not scale below a 1-segment window. So instead of a constant additive increase, the code uses a logarithmically scaled additive increase that slowly adapts the additive increase constant to the slow start threshold. Despite these quite radical changes, the diff is surprisingly small. The design and implementation is explained in [Ahmed19], which also includes evaluation results.

4. IANA Considerations

This specification contains no IANA considerations.

5. Security Considerations

Section 3.5 on scaling down to fractional windows discusses the tradeoff in becoming unresponsive at a minium window, which causes a queue to build (harm to self and to others) but protects oneself against other unresponsive flows (whether malicious or accidental).

This draft inherits the security considerations discussed in [I-D.ietf-tsvwg-ecn-l4s-id] and in the L4S architecture [I-D.ietf-tsvwg-l4s-arch]. In particular, the self-interest incentive to be responsive and minimize queuing delay, and protections against those interested in disrupting the low queuing delay of others.

6. Acknowledgements

THanks to Vidhi Goel for suggested improvements, including providing new text.

Bob Briscoe's contribution was part-funded by the Comcast Innovation Fund. The views expressed here are solely those of the authors.

7. Comments and Contributions Solicited (To be removed before Publication)

Comments and questions are encouraged and very welcome. They can be addressed to the IRTF Internet Congestion Control Research Group's mailing list <iccrg@irtf.org>, and/or to the authors via <draft-briscoe-iccrg-congestion-control@ietf.org>. Contributions of design ideas and/or code are also encouraged and welcome.

8. Contributors

The following contributed implementations and evaluations that validated and helped to improve this specification:

Olivier Tilmans <olivier.tilmans@nokia-bell-labs.com> of Nokia Bell Labs, Belgium, prepared and maintains the Linux implementation of TCP Prague.

Koen De Schepper <koen.de_schepper@nokia-bell-labs.com> of Nokia Bell Labs, Belgium, contributed to the Linux implementation of TCP Prague.

Joakim Misund <joakim.misund@gmail.com> of Uni Oslo, Norway, wrote the Linux paced chirping code.

Asad Sajjad Ahmed <me@asadsa.com>, Independent, Norway, wrote the Linux code that maintains a sub-packet window.

9. References

9.1. Normative References

[I-D.ietf-tcpm-accurate-ecn] Briscoe, B., Kühlewind, M., and R. Scheffenegger, "More Accurate ECN Feedback in TCP", Work in Progress, Internet-Draft, draft-ietf-tcpm-accurate-ecn-18, 22 March 2022, <https://datatracker.ietf.org/doc/html/draft-ietf-tcpm-accurate-ecn-18>.

[I-D.ietf-tsvwg-ecn-l4s-id] Schepper, K. D. and B. Briscoe, "Explicit Congestion Notification (ECN) Protocol for Very Low Queuing Delay (L4S)", Work in Progress, Internet-Draft, draft-ietf-tsvwg-ecn-l4s-id-25, 4 March 2022,

<https://datatracker.ietf.org/doc/html/draft-ietf-tsvwg-ecn-l4s-id-25>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC3168] Ramakrishnan, K., Floyd, S., and D. Black, "The Addition of Explicit Congestion Notification (ECN) to IP", RFC 3168, DOI 10.17487/RFC3168, September 2001, <https://www.rfc-editor.org/info/rfc3168>.

[RFC8311] Black, D., "Relaxing Restrictions on Explicit Congestion Notification (ECN) Experimentation", RFC 8311, DOI 10.17487/RFC8311, January 2018, <https://www.rfc-editor.org/info/rfc8311>.

9.2. Informative References

[Ahmed19] Ahmed, A.S., "Extending TCP for Low Round Trip Delay", Masters Thesis, Uni Oslo , August 2019, <https://www.duo.uio.no/handle/10852/70966>.

[ecn-fallback] Briscoe, B. and A.S. Ahmed, "TCP Prague Fall-back on Detection of a Classic ECN AQM", bobbriscoe.net Technical Report TR-BB-2019-002, April 2020, <https://arxiv.org/abs/1911.00710>.

[I-D.ietf-tcpm-generalized-ecn] Bagnulo, M. and B. Briscoe, "ECN++: Adding Explicit Congestion Notification (ECN) to TCP Control Packets", Work in Progress, Internet-Draft, draft-ietf-tcpm-generalized-ecn-09, 31 January 2022, <https://datatracker.ietf.org/doc/html/draft-ietf-tcpm-generalized-ecn-09>.

[I-D.ietf-tcpm-hystartplusplus] Balasubramanian, P., Huang, Y., and M. Olson, "HyStart++: Modified Slow Start for TCP", Work in Progress, Internet-Draft, draft-ietf-tcpm-hystartplusplus-05, 16 June 2022, <https://datatracker.ietf.org/doc/html/draft-ietf-tcpm-hystartplusplus-05>.

[I-D.ietf-tcpm-rfc8312bis] Xu, L., Ha, S., Rhee, I., Goel, V., and L. Eggert, "CUBIC for Fast and Long-Distance Networks", Work in Progress, Internet-Draft, draft-ietf-tcpm-rfc8312bis-08, 30 May 2022, <https://datatracker.ietf.org/doc/html/draft-ietf-tcpm-rfc8312bis-08>.

[I-D.ietf-tsvwg-aqm-dualq-coupled] Schepper, K. D., Briscoe, B., and G. White, "DualQ Coupled AQMs for Low Latency, Low Loss and Scalable Throughput (L4S)", Work in Progress, Internet-Draft, draft-ietf-tsvwg-aqm-dualq-coupled-23, 4 May 2022, <https://datatracker.ietf.org/doc/html/draft-ietf-tsvwg-aqm-dualq-coupled-23>.

[I-D.ietf-tsvwg-l4s-arch] Briscoe, B., Schepper, K. D., Bagnulo, M., and G. White, "Low Latency, Low Loss, Scalable Throughput (L4S) Internet Service: Architecture", Work in Progress, Internet-Draft, draft-ietf-tsvwg-l4s-arch-18, 7 July 2022, <https://datatracker.ietf.org/doc/html/draft-ietf-tsvwg-l4s-arch-18>.

[L4S-home] "L4S: Ultra-Low Queuing Delay for All", <https://riteproject.eu/dctth/#code>.

[linux-code] "Linux kernel tree with L4S patches", <https://github.com/L4STeam/linux>.

[LinuxPacedChirping] Misund, J. and B. Briscoe, "Paced Chirping - Rethinking TCP start-up", Proc. Linux Netdev 0x13 , March 2019, <https://www.netdevconf.org/0x13/session.html?talk-chirp>.

[patch-alpha-zero] Shewmaker, A. G., "tcp: allow dctcp alpha to drop to zero", Linux GitHub patch; Commit: c80dbe0, 23 October 2015, <https://github.com/torvalds/linux/commits/master/net/ipv4/tcp_dctcp.c>.

[patch-loss-react] De Schepper, K., "tcp: Ensure DCTCP reacts to losses", Linux GitHub patch; Commit: aecfde2, 4 April 2019, <https://github.com/torvalds/linux/commits/master/net/ipv4/tcp_dctcp.c>.

[PerAckEWMA] Briscoe, B., "Improving DCTCP/Prague Congestion Control Responsiveness", Technical Report TR-BB-2020-002, 20 January 2021, <https://arxiv.org/abs/2101.07727>.

[PragueLinux] Briscoe, B., De Schepper, K., Albisser, O., Misund, J., Tilmans, O., Kühlewind, M., and A.S. Ahmed, "Implementing the `TCP Prague' Requirements for Low Latency Low Loss Scalable Throughput (L4S)", Proc. Linux Netdev 0x13 , March 2019, <https://www.netdevconf.org/0x13/session.html?talk-tcp-prague-l4s>.

[RFC3649] Floyd, S., "HighSpeed TCP for Large Congestion Windows", RFC 3649, DOI 10.17487/RFC3649, December 2003, <https://www.rfc-editor.org/info/rfc3649>.

[RFC4340] Kohler, E., Handley, M., and S. Floyd, "Datagram Congestion Control Protocol (DCCP)", RFC 4340, DOI 10.17487/RFC4340, March 2006, <https://www.rfc-editor.org/info/rfc4340>.

[RFC4960] Stewart, R., Ed., "Stream Control Transmission Protocol", RFC 4960, DOI 10.17487/RFC4960, September 2007, <https://www.rfc-editor.org/info/rfc4960>.

[RFC5033] Floyd, S. and M. Allman, "Specifying New Congestion Control Algorithms", BCP 133, RFC 5033, DOI 10.17487/RFC5033, August 2007, <https://www.rfc-editor.org/info/rfc5033>.

[RFC5681] Allman, M., Paxson, V., and E. Blanton, "TCP Congestion Control", RFC 5681, DOI 10.17487/RFC5681, September 2009, <https://www.rfc-editor.org/info/rfc5681>.

[RFC6679] Westerlund, M., Johansson, I., Perkins, C., O'Hanlon, P., and K. Carlberg, "Explicit Congestion Notification (ECN) for RTP over UDP", RFC 6679, DOI 10.17487/RFC6679, August 2012, <https://www.rfc-editor.org/info/rfc6679>.

[RFC6928] Chu, J., Dukkipati, N., Cheng, Y., and M. Mathis, "Increasing TCP's Initial Window", RFC 6928, DOI 10.17487/RFC6928, April 2013, <https://www.rfc-editor.org/info/rfc6928>.

[RFC8257] Bensley, S., Thaler, D., Balasubramanian, P., Eggert, L., and G. Judd, "Data Center TCP (DCTCP): TCP Congestion Control for Data Centers", RFC 8257, DOI 10.17487/RFC8257, October 2017, <https://www.rfc-editor.org/info/rfc8257>.

[RFC8298] Johansson, I. and Z. Sarker, "Self-Clocked Rate Adaptation for Multimedia", RFC 8298, DOI 10.17487/RFC8298, December 2017, <https://www.rfc-editor.org/info/rfc8298>.

[RFC8888] Sarker, Z., Perkins, C., Singh, V., and M. Ramalho, "RTP Control Protocol (RTCP) Feedback for Congestion Control", RFC 8888, DOI 10.17487/RFC8888, January 2021, <https://www.rfc-editor.org/info/rfc8888>.

[RFC8985] Cheng, Y., Cardwell, N., Dukkipati, N., and P. Jha, "The RACK-TLP Loss Detection Algorithm for TCP", RFC 8985, DOI 10.17487/RFC8985, February 2021, <https://www.rfc-editor.org/info/rfc8985>.

[RFC9000]
Iyengar, J., Ed. and M. Thomson, Ed., "QUIC: A UDP-Based Multiplexed and Secure Transport", RFC 9000, DOI 10.17487/RFC9000, May 2021, <https://www.rfc-editor.org/info/rfc9000>.

[Tensions17] Briscoe, B. and K. De Schepper, "Resolving Tensions between Congestion Control Scaling Requirements", Simula Technical Report TR-CS-2016-001; arXiv:1904.07605, July 2017, <https://arxiv.org/abs/1904.07605>.

Authors' Addresses

Koen De Schepper
Nokia Bell Labs
Antwerp
Belgium

Email: koen.de_schepper@nokia.com
URI: https://www.bell-labs.com/usr/koen.de_schepper

Olivier Tilmans
Nokia Bell Labs
Antwerp
Belgium

Email: olivier.tilmans@nokia-bell-labs.com

Bob Briscoe (editor)
Independent
United Kingdom

Email: ietf@bobbriscoe.net
URI: http://bobbriscoe.net/