Improving DCTCP/Prague Congestion Control Responsiveness

Bob Briscoe*

07 Feb 2021

Abstract

This report explains how DCTCP introduces 1–2 rounds of unnecessary lag, due to the way it processes congestion feedback. To solve this, a per-ACK moving average is proposed. It always cuts out 1 RTT of this lag and, paradoxically, it cuts out most of the rest of the lag by spreading the congestion response over a round. The EWMA still averages out variations in the feedback signal over the same set number of round trips, even though it is clocked per-ACK. This version of the report is released prior to full evaluation, in order to elicit early feedback on the design.

1 Problem

This report shows that common implementations of DCTCP [AGM+10] (e.g. in Windows, Linux, or FreeBSD [BTB+17]) take up to two rounds before a change in congestion on the path fully feeds into the moving average that regulates its response. This is on top of the inherent round trip of delay in the feedback loop.

This applies equally to congestion controls based on DCTCP, in particular TCP Prague that is intended for use over the wide area; unnecessary rounds of lag will be much more noticeable in absolute terms. Unless otherwise stated, the term DCTCP in this paper should be considered to include derivatives like Prague.

A moving average intentionally dampens responsiveness so that there will only be a full response to a change if it sustains over the whole averaging period. However, the extra rounds we focus on here represent pure lag before that damping can even start.

This means that established DCTCP flows take 2 or 3 rounds (rather than 1) before they even start to respond to a reduction in available capacity or yield to a new flow. In turn, this means that a new flow must either build up a large queue before any established flow yields, or it must enter the system very tentatively. Therefore, the algorithm in this report is at least part of a solution to the 'Prague Requirement' for 'faster convergence at flow start' [DSBE20, Appx A.2.3].

Both extra rounds are due to DCTCP's two-stage process for responding to congestion (see Figure 1). The first stage introduces one round of delay (RTT_i) while it accumulates the marking fraction before it can calculate the EWMA (α_i). It passes this to the second stage that reduces the congestion window (W) by $\alpha_i W_i$. The second extra round arises because the second stage is triggered by congestion feedback (a red ACK) that occurs independently of the regularly clocked first stage.

So it takes two rounds before a full round of the congestion that triggered the start of the second stage has fed through into the EWMA that the first stage passes to the second. This is exacerbated by entering congestion window reduced (CWR) state at the first sign of congestion, which suppresses any further response for a round, just when more congestion feedback is likely. Also, the lagged congestion response will tend to overrun into the subsequent round, causing undershoot.

The problem seems to boil down to how to update the EWMA of marking on a per-ACK basis. But, more specifically, the problem is how to keep the time constant over which this per-ACK EWMA smooths itself to a set number of rounds, even though the number of packets per round varies.

2 Per-ACK EWMA

Instead of the EWMA of the marking probability being upscaled by a constant factor, it is proposed to upscale it by `flight_ / gain`, where `flight_` is the packets in flight, and `gain` is a constant (`gain < 1`). Then, as shown below, the EWMA can be maintained by a single continuous set of repetitive increments or decrements determined on each ACK.

Bob: Double-check that the use of `flight_` is rigorous.

*research@bobbriscoe.net,

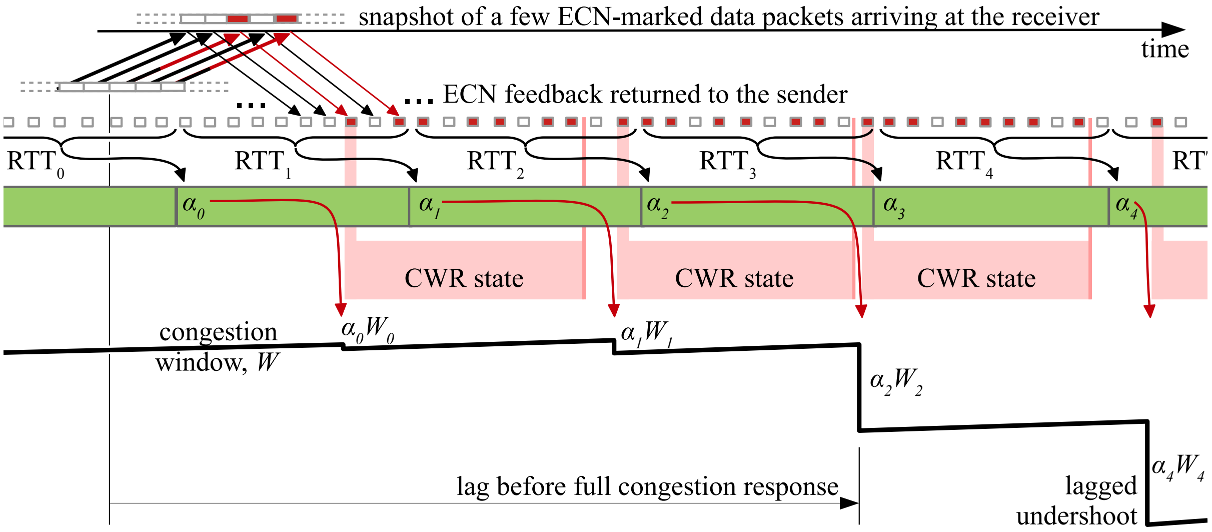

Figure 1: The problem: DCTCP's two stages for processing congestion feedback: 1) gathering feedback in a fixed sequence of rounds (RTT$_i$) to calculate the EWMA (α_i); 2) applying this EWMA on the first feedback mark, when it has had no time to gather enough feedback, which leads to a typically inadequate congestion response before entering congestion window reduced (CWR) state, which suppresses any further response for a round. See text for full commentary.

Although it updates every ACK, we show that scaling up the EWMA by `flight_` *implicitly* smooths the EWMA over a characteristic number of RTTs (specifically, `RTT/gain`), no matter how many ACKs there are per RTT. This contrasts with *explicitly* clocking per round that requires the two-stage process with its inherent extra lag.

The scaled up EWMA is effectively a smoothed count of the number of congestion marks per RTT, but scaled up by the constant `1/gain`. The number of marks per RTT (`v`) is related to the marking probability (`p`) by `v = p*flight_`.

Classical congestion controls suppress any further response for a round because their initial response is large and fixed. It seems wrong for DCTCP to mimic the timing of a classical congestion response, when it does not mimic its size. On first onset of congestion, DCTCP immediately responds with a tiny reduction (based on the previous absence of congestion), but then perversely it suppress any further response for a round trip.

So, once we have an EWMA of congestion marks that is updated continually on every ACK, it becomes possible to spread the reduction over the round. Then, if congestion continues to rise during the round, the EWMA will grow, and the response can pick up this growth as it proceeds.

Definitions of variables

`G` = `1/gain` (> 1). By default in DCTCP `G = 16`;

`av_up` : EWMA of the marks per round upscaled by `G`; Alternatively, it might help to think of this as the EWMA of the marking probability (alpha in DCTCP) upscaled by `G * flight_`.

`flight_` : the number of packets in flight when the marking probablity was fed into the EWMA (used for explanation, but not in the code);

`flight` : the number of packets in flight now;

`ce_fb` = 1 if ECN feedback per pkt; 0 otherwise.

2.1 Intuition

In DCTCP, the EWMA, `alpha`, is maintained per round trip as follows (in floating point arithmetic):

```
alpha += (F - alpha)/G,
```

where F is the fraction of marked bytes accumulated over the last round trip.

This can be approximated (see Appendix A) by repeatedly updating the EWMA on the feedback of every packet, but scaling down each update by the number of packets in that round, `flight_`. That is, the following per-packet update:

```
alpha += (ce_fb - alpha) / (flight_*G).
```

The above per-packet update of the EWMA is roughly equivalent to the following per-packet update of the upscaled EWMA, `av_up`:

```
av_up += ce_fb - av_up/(flight_*G).
```

2.2 Implementation

2.2.1 Maintaining the EWMA

The `ce_fb` term can be implemented by adding 1 to `av_up` on feedback of each CE-marked packet. If `av_up` was not upscaled by `G`, this would equivalent to adding `1/G` of a mark.

The number of packets acknowledged in the current round is `flight`. So repeatedly subtracting `av_up/(flight_*G)` on the arrival of every ACK would reduce `av_up` by `av_up*flight/(flight_*G)` in a round (see Appendix A). This approximates to `av_up/G` per round.

```
On_each_ACK'd_packet {
  // Update EWMA
  av_carry = div(
    av_carry.rem+av_up, flight*G);
  av_up += ce_fb - av_carry.quot;
}
```

The pseudocode uses an integer division function `div()` with the same interface as `div()` in C's standard library.[1] Specifically, it takes numerator and denominator as parameters and returns both the quotient and the remainder in the following structure:

```
typedef struct {
  int quot;
  int rem;
} div_t;
```

The variable `av_carry` would be declared of type `div_t`. It is used to carry forward the remainder to the invocation on the next ACK. The quotient will typically be either 0 or 1, which is then used to decrement the EWMA.

In practice, values would need to be checked for underflow and/or overflow (for instance, `av_up` has to be prevented from going negative in the last line), but such details are omitted from the pseudocode in this paper.

Figure 2 compares toy simulations of the above EWMA and the DCTCP EWMA (without changing cwnd). It can be seen that, whenever marks arrive, the algorithm always moves immediately, whereas DCTCP's EWMA does nothing until the next round trip cycle.

[1] `div()` can be thought of as a wrapper round `do_div()`, which is intended for kernel use, but less readable for pseudocode.

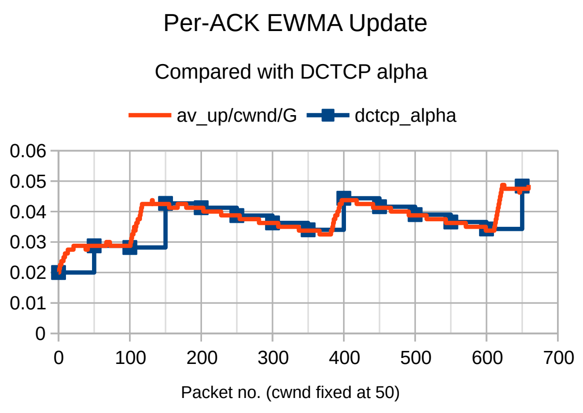

Figure 2: Initial verification of per-ACK EWMA algorithm (with constant `cwnd`).

Bob: Add discussion of initialization of the EWMA. To mimic DCTCP and Prague, on the first CE marked feedback, (av_up = cwnd * G). However, it will be interesting to see whether we can improve on this in a further set of experiments; for instance using (av_up = flight * G) instead

2.2.2 Responding to Congestion

The approach proposed in this section is not necessarily the final word on how to use the per-ACK EWMA for scalable congestion control (see § 6 for further ideas). Nonetheless, as a first step, we build incrementally on DCTCP, using its teaching selectively, but not departing too far from its intent.

DCTCP reduces `cwnd` by `alpha*cwnd/2` in any round trip in which CE feedback is present. The proposed approach reduces `cwnd` by half the average number of marked packets per round, or `av_up / (2*G)`. This is broadly equivalent to DCTCP in that it maintains the scalable '$1/p$' response function, except the following two slight differences could potentially alter the steady-state outcome:

- The window reduction is taken as a proportion of what the window has been in recent rounds, not what it is now (as DCTCP does) (see § 3).
- The window reduction is taken as a proportion of the amount of the window that has been *used* in recent rounds, not the maximum that the flow was entitled to use, i.e. packets in flight not `cwnd`. Thus, if an application-limited flow has only used a quarter of the available window in recent rounds, the proposed reduction of `cwnd` will be only a quarter of that applied previously by DCTCP (see § 3.3).

The main departure from DCTCP is in the speed of response. Rather than reduce cwnd on the first sign of CE feedback then suppress further response for a round trip, it is proposed to spread the reduction over the round following the first sign of CE feedback. In other words, use the whole round while in CWR state to reduce cwnd as the EWMA updates, such that, by the end of the round it will still have reduced as much as it would have done if the whole reduction had been applied at the end.

This will exploit the fact that the EWMA (av_up) is continually updated on every ACK. So, at one extreme, if the first CE mark is immediately followed by many others, the EWMA will rapidly increase early in the round of CWR, and cwnd can be rapidly decreased accordingly. While, at the other extreme, if the first CE mark is the only CE mark in the round, cwnd will still have reduced by alpha*cwnd/2 by the end of the round, but the EWMA will hardly have increased above the value it took when the CWR round started.

To spread the reduction over the round, the proposed algorithm below does not divide the round into an arbitrary number of points where cwnd is altered by varying amounts. Instead, it decrements cwnd on the first ACK when calculates that at least one packet of movement is possible.

```
On_each_ACK'd_packet {
  // Update EWMA
  av_carry = div(
    av_carry.rem+av_up, flight*G);
  av_up += ce_fb - av_carry.quot;

  if (!cwr && ce_fb) {
    // Record start of CWR state
    next_seq = snd_next;
    cwr = true;
  }
  if (cwr) {
    // Check still in CWR round
    if (snd_una < next_seq) {
      // Multiplicative Decrease
      cwnd_carry = div(
        cwnd_carry.rem+av_up, flight*G*2);
      cwnd -= cwnd_carry.quot;
    } else {
      cwr = false;
    }
  }
}
```

As in DCTCP, the EWMA continues to be calculated whether or not there is congestion feedback, and it is only used if there is actual congestion marking. However, unlike DCTCP, it continues to be applied per-ACK during the round of CWR rather than just once at the start of CWR.

CWR state then takes on a meaning that is nearly the opposite of its classical meaning. It no longer means 'congestion window reduced; no further reduction for a round'. Instead it means 'congestion window *reduction* in progress during this round'.

Although the motivation for this algorithm was not to prevent the stall caused by sudden decrease in cwnd, it would probably serve to address this problem as well. Therefore it should supplant proportional rate reduction (PRR [MD13]), at least when responding to ECN.

Details of cwnd processing are omitted from the pseudocode if they are peripheral to the proposed changes. For instance, in real code cwnd would be prevented from falling below a minimum (default 2 segments), and the slow-start threshold would track reductions in cwnd (but see § 6 for alternative ideas). Also, the code would have to handle acknowledgement of multiple packets at a time (potentially with different congestion markings). It might also handle ECN markings on packets of different sizes.

Also, for clarity, various integer arithmetic tricks are omitted from the pseudocode, such as choosing a value of G that is a power of 2, so that multiplication by G can be implemented with a bit-shift.

The pseudocode does, nonetheless, inherently attend to details such as loss of precision due to integer truncation. And note that cwnd is not updated on the ACK that ends CWR state, because it is updated on the marked ACK that starts CWR state.

2.2.3 The Whole AIMD Algorithm

```
On_each_ACK'd_packet {
  // Update EWMA
  av_carry =
    div(av_carry.rem+av_up, flight*G);
  av_up += ce_fb - av_carry.quot;

  if (!ce_fb) {
    // Additive Increase
    cwnd_carry = div2(
      cwnd_carry.rem[0]+G*2,
      flight*G*2, 0);
    cwnd += cwnd_carry.quot;
  } else if (!cwr) {
    // Record start of CWR state
    next_seq = snd_next;
    cwr = true;
  }

  if (cwr) {
    // Check still in CWR round
    if (snd_una < next_seq) {
      // Multiplicative Decrease
      cwnd_carry = div2(
        cwnd_carry.rem[1]+av_up,
        flight*G*2, 1);
      cwnd -= cwnd_carry.quot;
    } else {
      cwr = false;
    }
  }
}
```

Additive Increase: For completeness, the pseudocode above includes a Reno-like additive increase. This is intended for periods when the congestion control is close to its operating point (if it is not, see § 6).

Unlike DCTCP (but like Prague), the proposed AI algorithm is not suppressed during CWR state, with the following reasoning. DCTCP-like congestion controls are designed to induce roughly 2 ECN marks per round trip in steady state, so RTTs without marks are not meant to happen. Confining increases to periods that are not meant to happen creates an internal conflict within DCTCP's own design. Then, the algorithm's only escape is to store up enough decrease rounds to make space for a compensating period of increase. This has been found to cause unnecessary queue variation.

Instead, we continue additive increase regardless of CWR state. In place of suspending additive increase for a whole round, a fractional increase (which is calculated per-ACK as in most TCP implementations), is skipped if an ACK carries congestion feedback. This thins down the additive increase as congestion rises (see § 3.1 of [BDS17]).

Unlike DCTCP and unlike Prague, the increase in `cwnd` per round trip is divided over the actual number of packets in flight, not over the congestion window (which increases `cwnd` by 1 segment whether or not `cwnd` is fully used).

Shared Remainder The additive increase stores its remainder in the same `*cwnd_carry` variable as the multiplicative decrease. So both numerator and denominator are scaled up such that AI uses the same denominator parameter as MD, otherwise the upscaling of the carry variable would be different.

For both the AI and the MD, it will be noticed that a different division function, div2(), has been used to update `cwnd_carry`. This allows the remainder to be used in both directions without relying on signed integers.

Before the division for AI, the numerator is added to the previous remainder. But, for MD, the numerator needs to be subtracted from the previous remainder. Using signed integers would halve the available number space. Instead, the remainder is always kept positive in the range `[0, denom-1]`. This is achieved by maintaining two positive remainders in a two-element array, `rem[2]`. Then `rem[0]` is used as the remainder for an increase in `cwnd`, while `rem[1]` is used for a decrease.

This uses the division function, `div2()` defined below, which wraps the kernel macro `do_div()`, which is used for efficient integer division. `do_div()` already returns the positive remainder, and writes the result of the division into the numerator that it was called with. `div2()` returns a structure containing both remainders, also defined below. And it takes an extra boolean flip parameter to say which of the remainders `do_div` should write into.

```
typedef struct {
  int quot;
  int rem[2];
} div2_t;

struct div2_t div2(num, denom, flip)
{
  struct div2_t result;
  result.rem[flip] = do_div(num, denom);
  result.rem[!flip] = denom - result.rem[flip];
  result.quot = num;
  return result;
}
```

Whenever either remainder changes the other is kept in sync as its complement by the following relation: `rem[1] = denom - rem[0]`. For example, after a division by 1000, if the remainder is 400 for increases, the remainder for decreases will be 600. This can be thought of a jump from the bottom to the top of the denominator number space and a flip to viewing everything in the negative direction as if it was positive.

This ensures that the final result of a division (`quot`) is always rounded down (in the negative direction) before altering `cwnd`. Within any division that will be used for a decrease, the `denom` term added within `rem[1]` effectively rounds up, but only to keep everything positive within the division function—the result used outside the division function is still rounded down.

3 (Non-)Concerns

3.1 Circular Dependency?

There seems to be a circular dependency, because `av_up` is both upscaled by `flight_` then used to update `cwnd`, which determines `flight`.

In fact, `av_up` is upscaled by `flight_` (note the trailing underscore), which is what `flight` *was* at the time of each repetitive decrement or increment of `av_up`. So `av_up` depends on an implicit exponentially weighted moving average of `flight` with a characteristic smoothing timescale of `G` round trips. This removes any circular dependency.

3.2 Stale Remainder Scaling?

The denominator of each division used to calculate the remainder is not constant because it contains `flight`. Therefore, if `flight` is growing, a remainder calculated using earlier values of `flight` will be smaller than it ought to be. And vice versa if `flight` is shrinking.

This is a second order effect that should not alter the steady state that the AIMD converges to; it only puts a slightly different curve on the path to reach the steady state.

However, it is important to keep the two remainders in sync straight after either one has been updated. Otherwise if the complementary remainder is calculated later, the denominator could have changed, leading to possible subtle underflow bugs (or more complex code to catch these cases).

3.3 Advantage to Application Limited Flows?

The reduction to `cwnd` is spread over the `flight` packets in a round of CWR, so each reduction is scaled down by `flight` in the denominator of the call to `div()`. This means that `cwnd` is actually decreased by a multiplicative factor of `flight`, not of `cwnd`.

If a flow is not application-limited, the two amount to the same thing. But for app-limited flows, `flight` can be lower than `cwnd`, so the reduction in `cwnd` will be lower.

If a flow is only using a fraction of its congestion window, but it is still experiencing congestion, there is an implication that other flow(s) must have filled the capacity that the app-limited flow is 'entitled' to but not using. Then, it could be argued that the other flows have a higher `cwnd` than they are 'entitled' to, so that the app-limited flow can reduce its 'entitlement' (`cwnd`) less than these other flows in response to congestion.

If this argument is not convincing, the reduction in `cwnd` could be scaled up by `cwnd/flight`. However, it is believed that the code is reasonable, perhaps better, as it stands.

Similar arguments can be used to motivate additive increase over the actual number of packets in flight, rather than the potential congestion window.

4 Evaluation Plan

For research purposes, we ought not to introduce two changes at once, without evaluating each separately. Therefore, initially, we ought to use the continually updated EWMA, `av_up` to reduce `cwnd` in the classical way. That is, on the first feedback of a CE mark, reduce `cwnd` once by `av_up/(2*G)`. Then suppress further response for a round (CWR state). This should remove one round of lag (originally spent accumulating the marking fraction), but not the rest (spent reducing cwnd in response to a single mark, then doing nothing for a round while the extent of marking is becoming apparent).

On the same principle of one change at a time, A-B testing ought to use TCP Prague not DCTCP as the base for comparisons, given Prague includes fixes to known problems with DCTCP's EWMA.[2]But it might also be interesting to compare this with the pre-2019 implementation of DCTCP that might have accidentally been more responsive in many scenarios. This would be possible without winding all the code back to that date by simply setting a floor for `alpha` at 15/1024 in the Prague code.

The actual comparison should focus on how quickly a Prague flow in congestion avoidance can reduce in response to a newly arriving flow or a reduction in capacity.

Pacing should be enabled in both A and B tests. But, initially, segmentation offload should be disabled in both, to simplify interpretation of the results before enabling offload in both A and B tests together.

[2] 'Bugs' in the implementation of the EWMA in DCTCP have been fixed in TCP Prague. Prior to a 2015 patch [She15], the integer arithmetic for the EWMA in Linux floored at 15/1024, which ensured a minimum window reduction even after an extended period without congestion marking. That patch toggled the EWMA to zero whenever it tried to reduce `alpha` below that floor, and remained unresponsive until congestion was sufficient to toggle `alpha` back up to 16/1024. The TCP Prague reference implementation maintains the EWMA in an upscaled variant of `alpha`, as well as using higher precision and removing rounding bias.

Initially the same gain as DCTCP and Prague (1/16) ought to be used. But it is possible that the reason DCTCP's gain had to be so low was because of the 1–2 rounds of built in lag in the algorithm. Therefore, it will be interesting to see if the gain can be increased (from 1/16 to 1/8 or perhaps even 1/2 ought to be tried).

The thinking here is that a fixed amount of lag in a response is not the same as smoothing. Lag applies the same response but later. Smoothing spreads the response out, adding lag to the end of the response, but not to the start. Given every DCTCP flow's response to each change has been lagged by 1–2 rounds, it is possible that all flows have had to be smoothed more than necessary, in order to prevent the excessively lagged responses from causing over-reactions and oscillations.

Our motivation for improving DCTCP's responsiveness is to ensure established flows yield quickly when new flows are trying to enter the system, without having to build a queue. However, removing up to two rounds of unnecessary lag should also help to address the incast problem, at least for transfers that last longer than one round. Therefore, incast experiments could also be of interest.

It will also be necessary to check performance in the following cases that might expose poor approximations in the algorithm (relative to Prague):

1. When the packets in flight has been growing for some time;
2. . . . or shrinking for some time;
3. When the flow is application limited, with packets in flight varying wildly, rather than tracking the smoother evolution of `cwnd`.

Bob: Outcome of the evaluations to be added here.

5 Related Work

Reducing `cwnd` in one RTT by half of the marks per round trip (`av_up/2/G`) is similar but not the same as Relentless TCP [Mat09], which reduces `cwnd` by half a segment on feedback of each CE-marked packet. The difference is that `av_up` is a moving average, so it does not depend on the number of marks in any specific round, whereas the Relentless approach does. Relentless was designed for the classical approach with smoothing in the network, so it immediately applies a full congestion response without smoothing. In contrast, using the moving average implements the smoothing in the sender.

Like DCTCP, the per-ACK congestion response proposed in section 5.2 of [AJP11] maintains an EWMA of congestion marking probability, `alpha`. But, unlike DCTCP, it reduces `cwnd` by half of `alpha` (in units of packets) on feedback of each ECN mark. This is partway between Relentless and DCTCP, because it uses the smoothed average of marking, but it applies it more often in rounds with more marks. This still causes considerable jumpiness, because marks tend to be bunched into one round then clear for a few rounds, particularly with step-marking. In contrast, the approach proposed in the present paper limits the reduction within any one round to the averaged number of marks per round (as DCTCP itself does).

6 Ideas for Future Work

An EWMA of a queue-dependent signal is analogous to an integral controller. It filters out rapid variations in the queue that do not persist, but it also delays any response to variations that do persist. Faster control of dynamics should be possible by adding a proportional element, to create a proportional-integral (PI) controller within the sender's congestion control. The proportional element would augment any reduction to the congestion window dependent on the rate of increase in `av_up`.

Separately, it would be possible to use the per-ACK EWMA of marks per round (`av_up`) as a good indicator of whether a flow has lost its closed-loop control signal, for instance because another flow has left the bottleneck, or capacity has suddenly increased. A flow could then switch into a mode where it searches more widely for a new operating point, for instance using paced chirping [MB19, § 3]. To deem that the closed loop signal had significantly slowed, it might calculate the average distance between marks implied by the EWMA `av_up`, multiply this by a heuristic factor, then compare this with the number of packets since the last mark. Alternatively, it might detect when the EWMA of the marks per round had reduced below some absolute threshold (by definition, the marks per round of a scalable congestion control in steady state should be invariant for any flow rate).

References

[AGM+10] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitu Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data Center TCP (DCTCP). *Proc. ACM SIGCOMM'10, Computer Communication Review*, 40(4):63–74, October 2010.

[AJP11] Mohammad Alizadeh, Adel Javanmard, and Balaji Prabhakar. Analysis of DCTCP: Stability, Convergence, and Fairness. In *Proc. ACM SIGMETRICS'11*, 2011.

[BDS17] Bob Briscoe and Koen De Schepper. Resolving Tensions between Congestion Control Scaling Requirements. Technical Report TR-CS-2016-001; arXiv:1904.07605, Simula, July 2017.

[BTB+17] Stephen Bensley, Dave Thaler, Praveen Balasubramanian, Lars Eggert, and Glenn Judd. Data Center TCP (DCTCP): TCP Congestion Control for Data Centers. Request for Comments RFC8257, RFC Editor, October 2017.

[DSBE20] Koen De Schepper and Bob Briscoe (Ed.). Identifying Modified Explicit Congestion Notification (ECN) Semantics for Ultra-Low Queuing Delay (L4S). Internet Draft draft-ietf-tsvwg-ecn-l4s-id-12, Internet Engineering Task Force, November 2020. (Work in Progress).

[Mat09] Matt Mathis. Relentless Congestion Control. In *Proc. Int'l Wkshp on Protocols for Future, Large-scale & Diverse Network Transports (PFLDNeT'09)*, May 2009.

[MB19] Joakim Misund and Bob Briscoe. Paced Chirping - Rethinking TCP start-up. In *Proc. Netdev 0x13*, March 2019.

[MD13] Matt Mathis and Nandita Dukkipati. Proportional Rate Reduction for TCP. Request for Comments 6937, RFC Editor, May 2013.

[She15] Andrew G. Shewmaker. tcp: allow dctcp alpha to drop to zero. Linux GitHub patch; Commit: c80dbe0; https://github.com/torvalds/linux/commits/master/net/ipv4/tcp_dctcp.c, October 2015.

A Approximations

The per-ACK EWMA is not intended to mimic a per-RTT EWMA. Otherwise, the per-ACK EWMA would have to reach the same value by the end of the round, irrespective of whether markings arrived early or late in the round. It is more important for the EWMA to quickly accumulate any markings early in the round than it is to ensure that the EWMA reaches precisely the same value by the end of the round.

Neither is it important that a per-ACK EWMA decays at precisely the same rate as a per-round EWMA (assuming they both use the same gain). The gain is not precisely chosen, so if a per-ACK EWMA decays somewhat more slowly, it is unlikely to be critical to performance (if so, a higher gain value can be configured).

However, it *is* important that a per-ACK EWMA decays at about the same rate however many ACKs there are per round, although the decay rate does not have to be precisely the same.

The per-ACK approach uses the approximation that one reduction with gain $1/G$ is roughly equivalent to n repeated reductions with $1/n$ of the gain. Specifically, that $(1 - 1/nG)^n \approx 1 - 1/G$.

$$\begin{aligned}(1-1/nG)^n &= 1 + \frac{n}{-nG} + \frac{n(n-1)}{2(-nG)^2} + \dots \\ &= 1 - \frac{1}{G} + O\left(\frac{1}{G^2}\right) \\ &\approx 1 - \frac{1}{G}\end{aligned}$$

To quantify the error, we define the effective gain $(1/G')$ as the per-RTT gain that would give an equivalent reduction to multiple smaller per-ACK reductions using the original gain $(1/G)$. Numerically, we find that $G' \approx G + 1/2$ (see Figure 3).

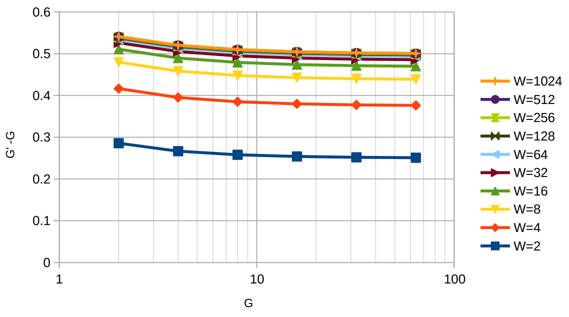

Figure 3: Difference between gain used for multiple per-ACK reductions, G', and the gain of one equivalent reduction, G.

For instance, multiple reductions with $G \approx 15.5$ are roughly equivalent to one reduction with $G' = 16$.

This can be explained because most of the error comes from omission of the $O(1/G^2)$ term. So we set

$$1 - \frac{1}{G'} \approx 1 - \frac{1}{G} + \frac{(n-1)}{2nG^2},$$

which, in the worst case of large n, reduces to

$$G' \approx \frac{2G^2}{(2G-1)}.$$

Then the difference between the reciprocals of the effective and actual gains is

$$G' - G \approx \frac{G}{(2G-1)}$$

Other than for low values of G this difference is indeed roughly 1/2.

The worst-case error occurs when G is small and W is large. Multiple reductions using any high value of W and the lowest practical value of $G(=2)$ would be equivalent to a single reduction using $G' \approx 2.54$ (i.e. the error in this worst-case is about 0.54).

Document history

Version	Date	Author	Details of change
00A	07 Nov 2020	Bob Briscoe	First draft.
00B	29 Nov 2020	Bob Briscoe	Added `cwnd` reduction and increase. Defined reusable function `repetitive_div()`. Corrected use of `cwnd` to `flight`, and distinguished current `flight`, from `flight_` when marks were averaged. Added abstract; schematic of problem; sections on evaluation plan, related work and future work; and appendix on approximations.
00C	02 Dec 2020	Bob Briscoe	Altered algorithms from hand-crafted repetitive_div() to div() in stdlib
01	19 Jan 2021	Bob Briscoe	Added CC to title, altered abstract, added motivation to intro and issued.
02	20 Jan 2021	Bob Briscoe	Improved abstract, intro & eval'n plan.
03A	07 Feb 2021	Bob Briscoe	More care distinguishing DCTCP/Prague variants. Defined `div2()` to fix root cause of remainder underflow. Explained shared remainders properly. Changed `g` terminology to `G`. Added todo notes incl. EMWA init.