Improving DCTCP/Prague Congestion Control Responsiveness

Bob Briscoe*

19 Jan 2021

Abstract

This report motivates and defines an improvement to the responsiveness of the Data Center TCP (DCTCP) algorithm. It explains how DCTCP introduces unnecessary lag equivalent to 1.5–2 RTTs, due to the way it processes congestion feedback. A per-ACK moving average is proposed that cuts out 1 RTT of this lag. Paradoxically, the rest of the lag is reduced by spreading the congestion response over a round. The EWMA is designed to still smooth over the same set number of round trips, even though it is clocked per-ACK. This version 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.

A moving average intentionally dampens responsiveness in order 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. They are also on top of the inherent round trip of delay in the feedback loop.

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 the 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.

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

*research@bobbriscoe.net,

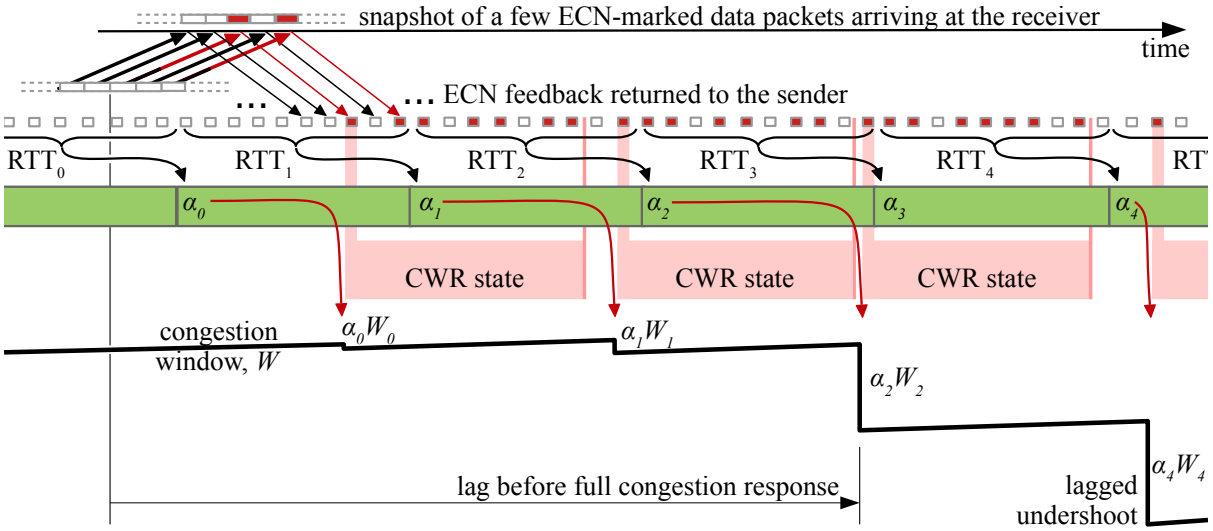

Figure 1: The problem: DCTCP's two stages for processing congestion feedback: 1) gathering feedback in a fixed sequence of rounds (RTT_i) to calculate the EWMA (α_i); 2) applying this EWMA on the first feedback mark, when it has had no time to gather enough feedback, which leads to a typically inadequate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

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`. 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. Given `av_up` is upscaled by `g`, this is 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 algorithm uses the integer division library function `div()`, which returns both the quotient and the remainder in a structure of type `div_t` as follows:

```
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.

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.

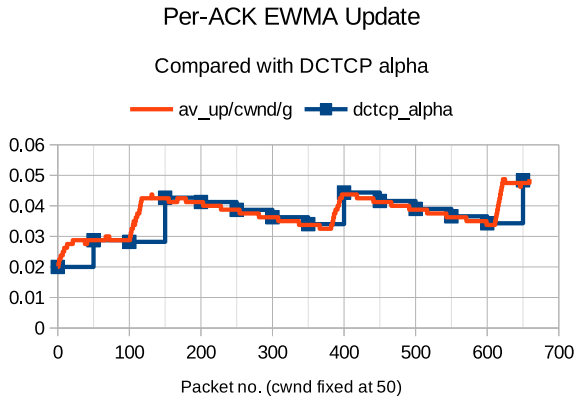

Figure 2: Initial verification of per-ACK EWMA algorithm (with constant `cwnd`).

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 in DCTCP) (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 by DCTCP (see § 3.2).

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` as soon as the algorithm first 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 is calculated continuously, but it is only used if there is actual congestion marking, when it is applied for one round 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 increase 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, but the Linux implementation of DCTCP works well enough without attending to this degree of detail.

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

For completeness, the pseudocode below 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).

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

  if (!ce_fb) {
    // Additive Increase
    cwnd_carry =
      div(g*2+cwnd_carry.rem, flight*g*2);
    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
      _denom = flight*g*2;
      cwnd_carry.rem =
        _denom - cwnd_carry.rem;
      cwnd_carry =
        div(cwnd_carry.rem+av_up, _denom);
      cwnd_carry.rem =
        _denom - cwnd_carry.rem;
      cwnd -= cwnd_carry.quot;
    } else {
      cwr = false;
    }
  }
}
```

Unlike DCTCP, the proposed algorithm does not suspend additive increase 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 increase 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 is calculated per-ACK (as in most TCP implementations) but it is skipped if an ACK carries congestion feedback. This thins down the additive increase as congestion rises (see § 3.1 of [BDS17]).

Also unlike DCTCP, the proposed algorithm increases `cwnd` by one segment over the actual window of packets in flight, not over the congestion window (which might not be fully used).

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.

Naively, the remainder could be added for AI and subtracted for MD. Instead, it is always added, but before and after the MD, it is respectively flipped to the opposite end of the number space of the denominator and back again. Whichever direction cwnd last moved in, this ensures that the remainder always lies in the range [0,`_denom`-1]. This is preferable to handling a negative carry, which would halve the number space available for the scaled up variables.

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 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 the decrease of `cwnd` is actually 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).

Initial experiments will need to compare how quickly a DCTCP flow in congestion avoidance can reduce in response to a newly arriving flow or a reduction in capacity.

Initially the same gain as DCTCP (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 two 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 by 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 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, it is possible that improved responsiveness will help address the incast problem, which will not have been helped by up to two rounds of unnecessary lag. Therefore, incast experiments could 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 DCTCP):

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-10, Internet Engineering Task Force, March 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.

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 $(1-1/ng)^n \approx 1-1/g$.

$$\begin{aligned}(1-1/ng)^n &= 1+\frac{n}{-ng}+\frac{n(n-1)}{2(-ng)^2}+\ldots\\ &= 1-\frac{1}{g}+O\left(\frac{1}{g^2}\right)\\ &\approx 1-\frac{1}{g}\end{aligned}$$

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).

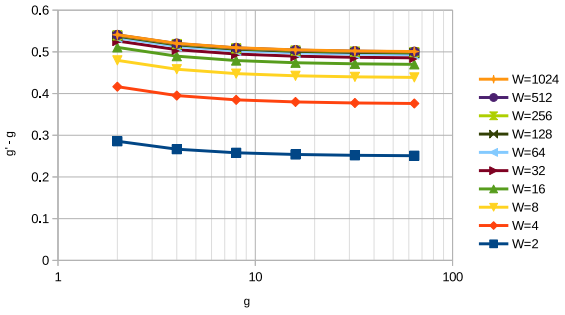

Figure 3: Difference between gain used for multiple per-ACK reductions, g', and the gain of one equivalent reduction, g.

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.

