Scaling TCP's Congestion Window for Small Round Trip Times

Bob Briscoe*

14 May 2015

Abstract

This memo explains that deploying active queue management (AQM) to counter bufferbloat will not prevent TCP from overriding the AQM and building large queues in a range of not uncommon scenarios. It is solely a paper study. The prevalence of these scenarios in practice will need to be established.

To keep its queue short, an AQM drops (or marks) packets to make the TCP flow(s) traversing it reduce their packet rate. Nearly all TCP implementations will not run at less than two packets per round trip time (RTT), and the TCP design cannot operate at less than one, at least not without significant modification. 2pkt / RTT need not imply low bit-rate if the RTT is small. For instance, it represents 2Mb/s over a 6ms round trip. When a few TCP flows share a link, in certain scenarios, including regular broadband and data centres, no matter how much the AQM signals to the flows to keep the queue short, they will not obey, because it is impossible for them to run below this floor. The memo proposes the necessary modification to the TCP standard.

1 The Problem

The capacity-seeking (aka. greedy) behaviour of TCP and its derivatives has led to the need for active queue management (AQM) which starts to drop packets as the queue grows, even when it is still quite short. Then the queue stays short, and the rest of the buffer remains available to absorb bursts.

Keeping down queuing delay, obviously drives down the round trip time (RTT). For a certain number of flows sharing a link, the packet rate of each will stay the same if the RTT reduces. But a lower RTT means less packets *per RTT*. Unfortunately, nearly all TCP implementations cannot operate at less than two packets per RTT (the standard [APB09] prohibits it).

How common are these circumstances? Imagine a quite unremarkable scenario in a residential broadband setting where 12 equal flows all share a 40 Mb/s link with an Ethernet frame size of 1518 B, so each sends at $40/12 = 3.3$ Mb/s. That's not so slow. But if an AQM attempts to keep their round trip time down to $R = 6\,ms$, they would have to run at $40M/(12 * 1518 * 8) * 6m = 1.64$ pkt/RTT. They cannot and will not do that.

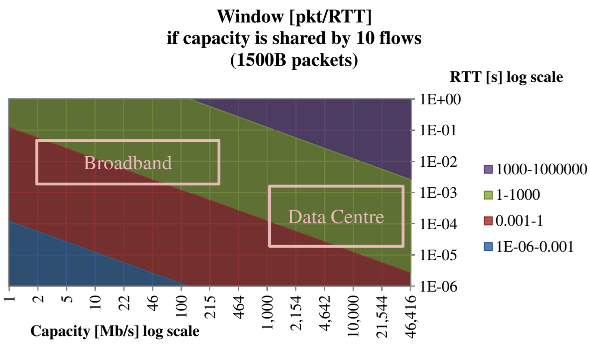

Figure 1: Window Size at Various Scales

A scenario with a shorter round trip time, a slower link, more flows or larger packets would require even less packets per round trip. In the developing world sub-packet windows are much more common [CISF11]. Nonetheless, taking an example scenario of 10 flows sharing the bottleneck, Figure 1 illustrates how the developed world is definitely not immune to the problem. Sub-single-packet windows are probably not at all unusual in both broadband and data centre scenarios.

TCP controls its rate using a window mechanism, where the window, W is the number of segments per round trip. The mechanism cannot work for a window of less than one segment, and TCP's standard congestion avoidance algorithm [APB09] stipulates a minimum window of 2*SMSS, where SMSS is the sender's maximum segment size (usually 1460 B). The 2 is presumably to work well with the common delayed ACK mechanism that defers an ACK until a second segment has arrived or the timer has expired (default 40 ms in Linux).

Once TCP's window is at this minimum, TCP no

*ietf@bobbriscoe.net, BT Research & Technology, B54/77, Adastral Park, Martlesham Heath, Ipswich, IP5 3RE, UK

longer slows down, no matter how much congestion signalling the AQM emits. TCP effectively ignores the increasingly insistent drops (or ECN marks) from the AQM. Inside the algorithm it halves the congestion window, but then rounds it back up to the minimum of two. For non-ECN flows, this will drive the AQM to make the queue longer, which in turn will drop more packets. So the flows will shuffle between periods waiting for timeouts and periods going faster than average while others wait for timeouts (see [Mor97]), but there will always be a longer queue. ECN flows will just keep making the queue longer until the RTT is big enough. In the following, where we don't need to distinguish ECN and non-ECN, the term 'signals' will be used for either drops or ECN marks.

As long as TCP effectively ignores congestion signals, queuing delay increases, the AQM emits even more signals, TCP's rounding-up effectively ignores them, the queuing delay increases, and so on. TCP is designed to reduce its window, not only when congestion signals increase, but also as RTT increases. So the queue will eventually stabilise at some larger size than the AQM would have liked (assuming there is sufficient buffer above the AQM's target queuing delay). Balance will be reached when all the flows are sending TCP's minimum number of segments per round trip. Because, above that, all the TCPs will reduce their window in response to any additional signals, but below that they won't.

So, that's good isn't it?

No. The flows are indeed sharing the link (without any losses in the case of ECN), they are clocking out packets twice every round trip and everything is stable. But to achieve this they have overridden the AQM to build a standing queue. A better outcome would be for all the TCPs to send out packets less often than twice (which ideally includes less then once) per round trip, which would keep the queue at the level intended by the AQM.

Note that this problem is not the same as TCP's "Silly Window Syndrome". Both problems do concern a sub-SMSS window, but the present problem concerns the congestion window, not the flow control window.

We thought that AQM was the solution to the queuing delay caused by TCP's capacity-seeking behaviour. However, in these scenarios TCP will trump AQM.

2 A Sub-MSS Window Mechanism for TCP Congestion Control

No amount of AQM twiddling can fix this. The solution has to fix TCP. TCP needs to be able to work internally with a fractional window instead of rounding it up to $2 * M$, where we use the symbol M for SMSS.

The window mechanism is fundamentally a way to send W bytes[1] every RTT, R.

We want to extend the window mechanism if $W < M$ to send a packet of M bytes every M/W round trips. Or more generally, send a packet of size s bytes every s/W round trips, where $s = \min(M, snd\_q)$, where snd_q is the amount of outstanding data waiting to be sent.

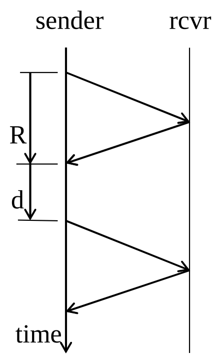

Figure 2: TCP's Segment Timing for $W < 1$

Normally, TCP holds back from sending if $W < s$. As illustrated in Figure 2, we need to modify this so that, following receipt of an ACK, if $W < s$ TCP waits for time d, where:

$$d + R = \frac{sR}{W}$$

$$d = \left(\frac{s}{W} - 1\right) R. \qquad (1)$$

Ironically, the sender has to insert a delay between each packet to avoid delay in the queue. Indeed, it is the same amount of delay. But it is better to localize the delay at each sender, not remote in the network, otherwise:

Mutual interest: There is a strong possibility that the remote queue is shared by other flows, some of which are likely to be interactive;

Self-interest: Many modern apps (e.g. HTTP/2, SPDY and interactive video) can adapt what

[1] Working in units of bytes not packets is necessary for what follows

they send and how much they send based on the local send queue, whereas it takes a round trip to know the status of a remote network queue;

TCP's basic window clocking machinery normally works as follows: when TCP sends s bytes, it decrements W by s and when the sender receives an acknowledgement for s bytes one RTT later, it increments W by s.

This needs to be modified as follows: after TCP's congestion response following receipt of an acknowledgement, if $W < s$, TCP must wait $(s/W - 1)R$, which then entitles it to send a packet of size s and decrement W by s, making W negative. The rest of TCP should then behave as usual.

Unless other parts of TCP have intervened in the meantime, TCP will receive the acknowledgement for s bytes and increment the window by s. W should now be positive again, but still insufficient to send a packet of size s. So TCP must again wait $(s/W - 1)R$.

The congestion control parts of TCP's machinery will be independently increasing W every time an ACK is received or reducing it every NACK. If the link becomes uncongested, as each packet is ACKed, TCP will increase W and consequently reduce the wait d between packets. Once $W \to s$ the wait $d \to 0$. Once $W > s$ no wait will be necessary. If instead the link remains congested, every time a NACK is received, W will reduce, and the wait between packets will increase appropriately.

The normal congestion responses of a set of TCP's with the above modification should work properly with the AQM at a bottleneck to pace the segments at less than one per round trip if necessary. Then they will balance with the AQM at its intended queuing delay, rather than bloating the queue just so they can all run at 2 segments per RTT.

This mechanism should be able to replace TCP's exponential back-off, as a more justifiable way to keep a congested link just busy enough during congestion. Every time TCP's retransmission timer expires, it will halve W, thus doubling the wait d before sending the next retransmission. With no response, W will get exponentially smaller and d exponentially larger. But as soon as there is one ACK, the window will grow so that a data packet (or probably a retransmission) can be sent.

ToDo: Integer arithmetic for Equation 1

3 Potential Issues

Even if d is large relative to R, TCP will have to use the last estimate of R because it will have no better way to estimate R given all activity will have stopped.

TCP's delayed ACK mechanism causes only every n (default 2) segments arriving at the receiver to elicit an ACK, unless more than the delayed ACK timer (default 40 ms in Linux) elapses between packets. Assuming the receiver delays ACKs, the above sub-MSS mechanism will result in all the segments being sent at the correct average rate, but in pairs. That is OK, but not ideal. The idea in AccECN [BSK14] where the sender can ask the receiver to turn off delayed ACKs would be nice.

4 Related Work

Morris [Mor97] found that much of the loss in the Internet in 1997 was due to many TCP flows at bottlenecks, causing an average window of less than one segment, which actually appears as a shuffling between some flows waiting for time-outs while others consume much more than the equal share. As a sign of the times, one proposed solution was to add more buffer space although it was recognised a more fundamental solution was really needed, for which RED was suggested, although as this memo points out, that would not have helped.

TCP Nice [VKD02] is intended for background transport. It is a modification to TCP Vegas to make it more sensitive to congestion and includes support for less than one segment in the congestion window. When the window is below 2 segments, it introduces a delay of s/W round trips between packets, rather than $(s/W - 1)$ as in the the above analysis. It arbitrarily limits the minimum window size to 1/48. The paper simulates Nice's interaction with the RED AQM.

Chen et al. [CISF11] investigates the behaviour of TCP when the path can only support a window of less than one segment, primarily interested in oversubscribed low capacity links in the developing world.

Komnios et al. [KSC14] find that LEDBAT performs better than TCP in the regime with a window of less than one segment, but when there is a mix of flows on the link, LEDBAT switches into its TCP mode, so it needs to be more sophisticated when it can do better by remaining in LEDBAT mode.

Acknowledgements

Thanks to Koen de Schepper who noticed and articulated the problem, to Michael Welzl for explaining why the minimum ssthresh in the TCP spec is $2 * \text{SMSS}$, and to Michael Welzl and Anna Brünstrom for helping with the Related Work. The ideas in Laminar TCP [Mat12] made it easier to work out a solution, by ignoring the congestion control part of TCP, and focusing solely on the part that keeps TCP 'clocking' at a constant rate.

The author was part-funded by the European Community under its Seventh Framework Programme through the Reducing Internet Transport Latency (RITE) project (ICT-317700). The views expressed are solely those of the author.

References

[APB09] M. Allman, V. Paxson, and E. Blanton. TCP Congestion Control. Request for Comments 5681, Internet Engineering Task Force, September 2009.

[BSK14] Bob Briscoe, Richard Scheffenegger, and Mirja Kühlewind. More Accurate ECN Feedback in TCP. Internet Draft draft-kuehlewind-tcpm-accurate-ecn-03, Internet Engineering Task Force, July 2014. (Work in Progress).

[CISF11] Jay Chen, Janardhan Iyengar, Lakshminarayanan Subramanian, and Bryan Ford. TCP Behavior in Sub-Packet Regimes. In *Proc. SIGMETRICS11*, pages 157–158. ACM, June 2011.

[KSC14] I. Komnios, A. Sathiaseelan, and J. Crowcroft. Ledbat performance in sub-packet regimes. In *Wireless On-demand Network Systems and Services (WONS), 2014 11th Annual Conference on*, pages 154–161, April 2014.

[Mat12] Matt Mathis. Laminar TCP and the case for refactoring TCP congestion control. Internet Draft draft-mathis-tcpm-tcp-laminar-01, Internet Engineering Task Force, July 2012. (Work in progress).

[Mor97] R. Morris. TCP Behavior with Many Flows. In *Proceedings of the 1997 International Conference on Network Protocols (ICNP '97)*, ICNP '97, pages 205–, Washington, DC, USA, 1997. IEEE Computer Society.

[VKD02] Arun Venkataramani, Ravi Kokku, and Mike Dahlin. TCP Nice: A Mechanism for Background Transfers. *SIGOPS Oper. Syst. Rev.*, 36(SI):329–343, December 2002.

Document history

Version	Date	Author	Details of change
00A	15 May 2015	Bob Briscoe	First Draft
00B	14 May 2015	Bob Briscoe	Added Abstract, Scenarios and Related Work