Sweet TCP: Battling the Bottleneck Bufferbloat

Jaume Barcelo

***Abstract*—The transfer control protocol is designed to fill the *pipe* between the source and the destination for an efficient use of network resources. As a side effect, TCP also fills in the bottleneck buffer, which is the buffer that precedes the slowest link in the path. Permanently filling the buffer does not offer any performance advantage. On the contrary, permanently full buffers impair the operation of TCP causing excessive delays and timeouts. This problem is known in the literature as bufferbloat. In this paper, we make use of some of the insights developed by the community around the bufferbloat problem to propose a TCP congestion avoidance protocol that fills in the pipe but not the buffer. By continuously monitoring improvements and losses in terms of throughput and delay, Sweet TCP finds and stabilizes around the optimum operation point that simultaneously maximizes the throughput and minimizes the delay. A simple algorithm that adjusts the contention window is proposed. In contrast with traditional congestion avoidance behaviour, the proposed algorithm reduces the congestion window as soon as symptoms of bufferbloat are detected, and increases the congestion window as soon as the symptoms disappear.**

***Index Terms*—TCP, congestion avoidance, bufferbloat, delay**

I. Introduction

Many Internet communications use the Transfer Control Protocol (TCP). Fig. 1 is a copy from [1] and explains the behaviour of TCP as a function of the number of packets in-flight. The number of packets in-flight is closely related to the TCP congestion window ($CWND$) which limits the number of bytes in-flight. If the number of packets in-flight is too low, the link is not efficiently used. If it is too high, the packets accumulate in the buffer preceding the bottleneck link and the end-to-end delay increases.

In the figure we can observe that the behaviour of TCP presents a sweet point in which throughput is maximised and delay is minimised. If the current operation point is to the left of the optimal, the congestion window should be increased. If the current operation point is to the right of the optimal, the congestion window should be decreased. We try to devise a simple mechanism that can accomplish this goal.

The assumption is that every round-trip-time (RTT) we have access to an estimation of the delay and throughput and should take a decision about whether the congestion window should be increased or decreased. Using a single sample of delay and throughput it is difficult to decide whether TCP is operating at the left or the right of the sweet point. Therefore the strategy is to move along the horizontal axis in the figure by increasing or decreasing the number of packets in-flight, and then decide whether we are moving in the right direction or not.

In a completely ideal situation in which the TCP behaviour was exactly as represented in the figure, an increase in throughput or a decrease in delay would mean a move in the right direction. A decrease in throughput or an increase in delay would mean a move in the wrong direction. In a real dynamic network, it is not realistic to expect a behaviour that exactly mimics the idealized Fig. 1. For this reason we propose an algorithm that takes into account the relative variation of both delay and throughput in an attempt to make the right decision even when the measurements are noisy.

II. Operating in TCP's Sweet Point

The current TCP behaviour is to increase the number of packets in-flight as long as no packet loss occurs. Roughly speaking, the number of packets in-flight increases by one every RTT. This means that, unless Random Early Detection is used, TCP has a tendency to fill the buffer that precedes the bottleneck link and therefore to unnecessarily increase the delay of all the flows that traverse such link.

The problem is that correctly configuring RED is not trivial. If RED is too aggressive, TCP slows down to a point in which the link is not efficiently used. If RED is too quiescent, the queue builds up and the delays are too high.

We propose changing the mechanism that adjusts the congestion window mechanism in the congestion avoidance phase of TCP. Instead of blindly growing the congestion window until a packet loss occurs, we suggest to increase the congestion as long as it results in performance benefits. In other words, assuming that TCP starts at some point to the left of the optimal, the idea is to move to the right towards the sweet point.

While the point of operation is moved to the right by increasing the congestion window, the values of throughput and delay are monitored every RTT. When a performance loss is detected, probably in the form of same throughput and higher delay, TCP will start decreasing the congestion window. This is, when the sweet point has been crossed and moving to the right only increases the delay and does not offer throughput benefits, we start moving to the left. If this decrease results in a performance gain, probably in the form of same throughput and lower delay, TCP will continue decreasing the congestion window. It will keep decreasing it as long as this behaviour results in a performance gain. When decreasing the window results in a performance loss, probably in the form of lower throughput and same delay, TCP will revert to increasing. This mechanism will force TCP to operate around the sweet point where the throughput is maximum and the delay minimum.

The idea is to take measures of throughput and delay every RTT. We name these measures T_i and D_i where T stands for throughput and D for delay. The integer i is an index that increases every RTT. Note that TCP already computes an estimation of the RTT which can be used for D_i. To estimate T_i it would be necessary to compute the amount of acknowledged data during the RTT.

The authors are with ...

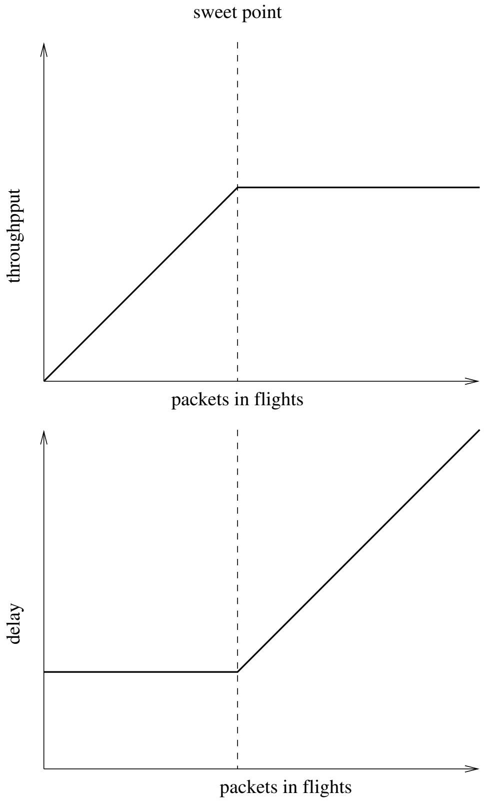

Fig. 1. Typical delay and throughput curves as a function of the number of packets in-flight. This number is closely related to the congestion window size $CWND$.

After obtaining the measures T_i and D_i, we compute the relative variation of throughput and delay as $T_{rv} = \frac{T_i - T_{i-1}}{T_{i-1}}$ and $D_{rv} = \frac{D_i - D_{i-1}}{D_{i-1}}$, respectively. Finally, we measure the performance improvement as $PEIM = T_{rv} - D_{rv}$.

Note that if the current operation point is at the left of the dashed line in Fig. 1 and we increase the congestion window (we move to the right), the throughput increases and the delay stays the same. Therefore we measure a positive performance improvement. Similarly, if the current operation point is at the right of the dashed line and we reduce the congestion window (we move to the left), the throughput stays the same and the delay decreases. Therefore we also measure a positive performance improvement. The performance improvement is positive whenever we move closer to the optimal operation point. Conversely, the performance improvement is negative whenever we move away from the operation point.

We can use these properties to control whether TCP should increase or decrease the congestion window. Basically, we should keep moving in the same direction when we measure a positive performance improvement $PEIM$ and change the direction otherwise.

The behaviour of the current TCP is $CWND_i = CWND_{i-1} + MSS$ every RTT whenever there is no packet loss. TCP always increases $CWND$ as long as there is no packet loss.

We suggest to update $CWND$ as $CWND_i = CWND_{i-1} + STAT_i * MSS$ where $STAT_i$ stands for state and can be 1 or -1. The $CWND$ increases when $STAT_i$ is positive and decreases when $STAT_i$ is negative. $STAT$ keeps the same value as long as there are performance improvements, and changes its value when there is a performance loss: $STAT_i = sign(PEIM) * STAT_{i-1}$, where $sign(\cdot)$ is the sign operator that is 1 for a positive value and -1 for a negative value . The goal is to keep moving on the same direction along the horizontal axis of the figure while there is a performance improvement and change the direction as soon as we detect a performance loss.

The following algorithm summarizes all the steps necessary to update $CWND$.

```
Entering congestion avoidance ...
$STAT_0 \leftarrow 1$
Measure $T_0$ and $D_0$
$i \leftarrow 1$
/* The following loop is executed once every RTT */
while no packet loss do
  Measure $T_i$ and $D_i$
  Compute $T_{rv} = \frac{T_i - T_{i-1}}{T_{i-1}}$
  Compute $D_{rv} = \frac{D_i - D_{i-1}}{D_{i-1}}$
  Compute $PEIM = T_{rv} - D_{rv}$
  Compute $STAT_i = sign(PEIM) * STAT_{i-1}$
  Compute $CWND_i = CWND_{i-1} + STAT_i * MSS$
  $i \leftarrow i + 1$
end while
```

III. Slow Start

The exponential growth of slow start should be maintained as long as the performance improvement stays positive. A negative performance improvement should move TCP to congestion avoidance.

IV. Packet Loss

Sweet TCP should react to packet loss just like current TCP implementations. The congestion window should be halved in the case of fast recovery and TCP should move back to the slow start in case of timeout.

V. Conclusion

Sweet TCP increases and decreases the congestion window while in the congestion avoidance mode to try to reach the point in which the throughput is maximised and the delay is minimized. This mechanisms avoids that bottleneck buffer is completely filled. In order to decide when to increase or decrease the congestion window, TCP has to keep track of the delay and throughput measured every RTT.

Acknowledgment

Thanks to Adria Villalba for suggesting that a possible solution for bufferbloat was to move the operation of TCP to the sweet point.

References

[1] J. Gettys and K. Nichols, “Bufferbloat: Dark buffers in the internet,” *Queue*, vol. 9, no. 11, p. 40, 2011.