Start Me Up: Determining and Sharing TCP's Initial Congestion Window

Safiqul Islam and Michael Welzl
Department of Informatics, University of Oslo, Norway
{safiquli | michawe}@ifi.uio.no

ABSTRACT

When multiple TCP connections are used between the same host pair, they often share a common bottleneck – especially when they are encapsulated together, e.g. in VPN scenarios. Then, all connections after the first should not have to guess the right initial value for the congestion window, but rather get the appropriate value from other connections. This allows short flows to complete much faster – but it can also lead to large bursts that cause problems on their own. Prior work used timer-based pacing methods to alleviate this problem; we introduce a new algorithm that "paces" packets by instead correctly maintaining the ACK clock, and show its positive impact in combination with a previously presented congestion coupling algorithm.

CCS Concepts

•**Networks** → **Transport protocols;**

Keywords

TCP pacing; coupled congestion control

1. INTRODUCTION

Finding a suitable initial congestion window (cwnd) has been debated for many years in the IETF. A large Initial Window (IW) can be very beneficial [5], yet problematic for low bandwidth links, e.g. in developing countries (although RFC 6928 [4] discusses a study involving South America and Africa, this has been criticized for focusing on the flows that used the proposed larger initial window instead of measuring the impact on competing traffic). This is a difficult engineering trade-off because TCP normally assumes no prior knowledge about the path when it applies the IW.

Often, concurrent TCP connections are used between the same source-destination pair. They can share a network bottleneck, in particular when they are in a tunnel, e.g. in case of a VPN. In such cases, it would be possible for newly joining flows to either use a cached connection state or a share of an aggregate from the ongoing transfers instead of "blindly" applying a constant value. RFC 2140 [11] describes these two cases as temporal and ensemble sharing; this paper is concerned with the latter case.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2016 ACM. ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

2. BACKGROUND

Each TCP connection maintains states (e.g., local process states, RTT, cwnd, ssthresh) in a data structure called Transport Control Block (TCB). Sharing TCB data across parallel TCP connections can improve transient performance during the initialization phase [11]. Ensemble-TCP (E-TCP) [6] expanded the idea of (ensemble) sharing of TCB data across parallel TCP connections in order to allow active connections to continuously benefit from each other. In E-TCP, the aggregate of an ensemble is no more aggressive than one TCP Reno connection. Ensemble Flow Congestion Management (EFCM) [9] extended E-TCP by allowing the aggregate of an ensemble to be as aggressive as n TCP connections. These two mechanisms both realize a service of joint congestion control, somewhat similar to the Congestion Manager (CM) [2] or when using application stream multiplexing as in e.g. SCTP [10] or QUIC [8] – but such sharing is easier to implement than the former and does not require application involvement as in the latter.

We have recently complemented E-TCP and EFCM with an algorithm that addresses some problems that both of these mechanisms have,[1] as well as a possible encapsulation scheme to ensure that connections traverse a common bottlenecks, in an Internet-draft [13]. The coupling algorithm in [13] is inspired by our prior work on coupling for media flows in the context of WebRTC / RMCAT [7, 12].

Sharing TCB data can be particularly beneficial for short flows (e.g., web on/off traffic); short flows joining an aggregate can significantly reduce their completion time due to acquiring a share of potentially large cwnd from active connections. The crux of such sharing/initialization is that it can create sudden bursts in the network, potentially leading to queue growth and packet loss. E-TCP and EFCM acknowledged this problem, and addressed it using pacing. Most pacing methods (including the ones proposed for E-TCP and EFCM) use timers to clock out packets at regular

[1]TCP congestion control is stateful, but these states are not addressed in the E-TCP and EFCM algorithms. For example, slow start after a retransmission timeout (RTO) should not happen on one flow while ACKs still arrive on another flow. Also, to emulate the backoff of a single flow, TCP's concept of loss events should be retained for the aggregate, meaning that there should be only one backoff irrespective of the number of losses within the same loss "round".

Algorithm 1 Ack clocking for new connection c, running as a replacement of any other connection's cwnd increase. Initially, number_of_acks_c = 0 and clocked_cwnd_c is c's target cwnd calculated by the algorithm in [13]. N = number of flows.

```
1: if clocked_cwnd_c <= 0 then
2:     return                      ▷ alg. ends; other connections
3:                                 ▷ can increase cwnd again
4: end if
5: if number_of_acks_c % N = 0 then ’
6:     send a new segment for connection c
7:     clocked_cwnd_c ← clocked_cwnd_c - 1
8: end if
9: number_of_acks_c ← number_of_acks_c + 1
```

intervals over an RTT. This is not without problems, both in terms of implementation in the end host as well as (somewhat counter-intuitively) the impact inside the network [1].

The algorithm in [13] does not include any form of pacing, and therefore produces bursts that can lead to the problems described above (potentially diminished by burst limitation mechanisms underneath, e.g. in Linux [3]). We supply a mechanism to avoid sudden bursts in this paper. Different from prior work, this mechanism does not rely on a timer but simply maintains the ACK clock of TCP, thereby minimizing the impact on the dynamics in the network.

3. DESIGN

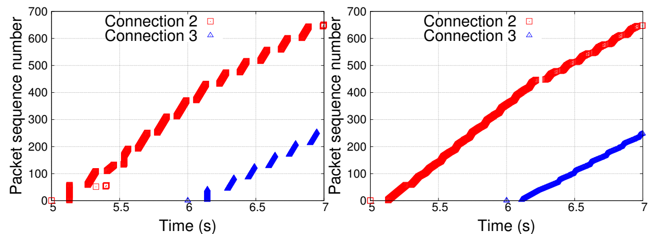
We begin by showing what happens when a new flow gets a share of a large aggregate with our mechanism in [13]; we simulate the behavior of three TCP connections in the ns-2 simulator[2] with a dumbbell topology (bottleneck capacity 10 Mbps, RTT 100 ms, packet size 1500 bytes, and queue length of 1 BDP (83 packets)). Connections 2 and 3 join after 5 and 6 seconds, respectively, and they receive large cwnd values from the aggregate. Figure 1(a) shows a time-sequence plot of connections 2 and 3. The congestion spike due to sudden bursts from connection 2 causes significant packet losses. Appropriate mixing of the two coupled flows did not play out well until the 3rd RTT for connection 2. A small burst is also visible when connection 3 joins.

We propose a simple mechanism to avoid these bursts. Rather than using timers, we make use of the ACKs connection 1 receives to clock packet transmissions of connection 2 over the course of the first RTT when connection 2 joins. Similarly, we make use of the ACKs of connections 1 and 2 to clock packet transmissions of connection 3. In this way, we avoid causing a burst in the network. Figure 1(b) illustrates that using the ack clock from the preexisting connection eliminates the congestion that is shown in Fig. 1(a).

When a connection c joins, it turns on the ack-clock feature and calculates the share of the aggregate, clocked_cwnd_c. Algorithm 1 illustrates the ack-clock mechanism that is used to distribute the share of the cwnd based on the acknowledgements received from other flows.

Figure 2 demonstrates the reduction of a short flow's completion times by immediately taking a share of the aggregate. This simulation was repeated 10 times with randomly picked flow start times over the first second for the long flow (25 Mb) and the sixth second for the short flow (200 Kb). We

[2]We used the TCP-Linux module that allowed us to use TCP code from Linux kernel (3.17.4) in simulations.



(a) Time-seq. diagram of connections 2 and 3, no ack-clock

(b) Time-seq. diagram of connections 2 and 3, ack-clocked

Figure 1: Coupling of 3 connections when connections 2 and 3 join after 5 and 6 seconds

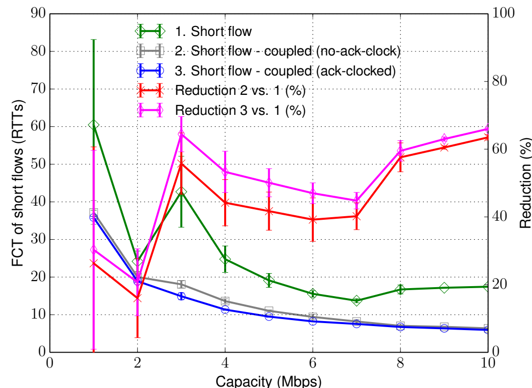


Figure 2: Flow completion time (FCT) of short flows.

used a dumbbell topology (RTT 100 ms, MTU 1500 bytes, queue length 1 BDP) while varying the capacity from 1 Mbps to 10 Mbps. It can be seen from Fig. 2 that there is a significant improvement in the short flow's completion time using our ack-clock mechanism, and the FCT is reduced by more than 40% for all other bottleneck capacities except 1 and 2 Mbps. The reduced competition also makes the behavior more predictable: the dip at 2 Mbps only exists when flows compete (here, the queue had just enough space for one, but not two flows each sending their Initial Window (IW)).

Because the long flow gets to rapidly increase its cwnd when a short flow terminates, the ack-clock mechanism reduced the FCT of the long flow too, but only by a negligible amount: only 0.66% or less in all cases.

4. CONCLUSION

We have presented an extension of our TCP congestion control coupling algorithm in [13] to maintain ACK clocking for multiple flows as if they were only a single flow. This allows to let newly starting flows of an aggregate quickly reap the benefit of an already large congestion window, reducing the flow completion times of short flows without incurring disadvantages of timer-based pacing methods.

We have not discussed what happens when another flow joins while this ACK clocking algorithm is active. This requires a slight extension of the algorithm that we will tackle in future work, together with other extensions of the algorithm in [13], e.g. to correctly handle quiescent senders. After a few such updates, we are confident that this algorithm will work significantly better than multiple competing TCPs in all possible cases, such that it would simply be a mistake to leave TCP connections uncoupled in situations where they are already encapsulated together (e.g. VPNs).

5. ACKNOWLEDGMENTS

This work has received funding from Huawei Technologies Co., Ltd., and the European Union's Horizon 2020 research and innovation programme under grant agreement No. 644334 (NEAT). The views expressed are solely those of the authors.

6. REFERENCES

[1] A. Aggarwal, S. Savage, and T. Anderson. Understanding the performance of TCP pacing. In *INFOCOM 2000. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 3, pages 1157–1165 vol.3, Mar 2000.

[2] H. Balakrishnan, H. Rahul, and S. Seshan. An integrated congestion manager architecture for internet hosts. In *Proc. ACM SIGCOMM*, 1999.

[3] Y. Cheng. Recent advancements in Linux TCP congestion control. IETF 88, Vancouver, 2013.

[4] J. Chu, N. Dukkipati, Y. Cheng, and M. Mathis. Increasing TCP's Initial Window. RFC 6928 (Experimental), Apr. 2013.

[5] N. Dukkipati, T. Refice, Y. Cheng, J. Chu, T. Herbert, A. Agarwal, A. Jain, and N. Sutin. An argument for increasing TCP's initial congestion window. *ACM SIGCOMM Computer Communications Review*, 40:27–33, 2010.

[6] L. Eggert, J. Heidemann, and J. Touch. Effects of ensemble TCP. *USC/Information Sciences Institute*, 7(1), December 1999.

[7] S. Islam, M. Welzl, S. Gjessing, and N. Khademi. Coupled congestion control for RTP media. *SIGCOMM Comput. Commun. Rev.*, 44(4):–, Aug. 2014.

[8] J. Iyengar, I. Swett, R. Hamilton, and A. Wilk. QUIC: A UDP-Based Secure and Reliable Transport for HTTP/2. I-D draft-tsvwg-quic-protocol-02, IETF, Jan. 2016. Work in Progress.

[9] M. Savorić, H. Karl, M. Schläger, T. Poschwatta, and A. Wolisz. Analysis and performance evaluation of the EFCM common congestion controller for TCP connections. *Computer Networks*, 49(2):269–294, 2005.

[10] R. Stewart. Stream Control Transmission Protocol. RFC 4960 (Proposed Standard), Sept. 2007. Updated by RFCs 6096, 6335, 7053.

[11] J. Touch. TCP Control Block Interdependence. RFC 2140 (Informational), Apr. 1997.

[12] M. Welzl, S. Islam, and S. Gjessing. Coupled congestion control for RTP media. Internet Draft draft-ietf-rmcat-coupled-cc, work in progress, Apr. 2016.

[13] M. Welzl, S. Islam, K. Hiorth, and J. You. TCP in UDP. Internet-Draft draft-welzl-irtf-iccrg-tcp-in-udp-00, Internet Engineering Task Force, Mar. 2016. Work in Progress.