How to Control a TCP: Minimally-Invasive Congestion Management for Datacenters

Safiqul Islam, Michael Welzl, Stein Gjessing
University of Oslo
Email: {safiquli, michawe, steing}@ifi.uio.no

***Abstract*—In multi-tenant datacenters, the hardware may be homogeneous but the traffic often is not. For instance, customers who pay an equal amount of money can get an unequal share of the bottleneck capacity when they do not open the same number of TCP connections. To address this problem, several recent proposals try to manipulate the traffic that TCP sends from the VMs. VCC and AC/DC are two new mechanisms that let the hypervisor control traffic by influencing the TCP receiver window (rwnd). This avoids changing the guest OS, but has limitations (it is not possible to make TCP increase its rate faster than it normally would). Seawall, on the other hand, completely rewrites TCP's congestion control, achieving fairness but requiring significant changes to both the hypervisor and the guest OS. There seems to be a need for a middle ground: a method to control TCP's sending rate without requiring a complete redesign of its congestion control. We introduce a minimally-invasive solution that is flexible enough to cater for needs ranging from weighted fairness in multi-tenant datacenters to potentially offering Internet-wide benefits from reduced inter-flow competition.**

I. Introduction

Datacenters have become a cornerstone of today's networked IT infrastucture. When the owner of a datacenter controls all communicating endpoints, a wide range of congestion control or traffic management mechanisms can be employed without having to worry about backward compatibility. Examples of such mechanisms are DCTCP [1], TIMELY [2], EyeQ [3], HyGenICC [4], Oktopus [5], SecondNet [6], and FairCloud [7].

In multi-tenant datacenters, however, the guest OSes of clients may be diverse and utilize an Internet-like mix of old and new TCP congestion control implementations, with and without ECN, following Reno, Cubic, BBR or any other TCP "flavor" that e.g. Linux and FreeBSD allow to configure via their pluggable congestion control frameworks [8]. This may put some users at a disadvantage, depending on how aggressively their congestion control probes for the available capacity. Moreover, unfair users may have an incentive to obtain a larger share of the capacity by opening multiple TCP connections. This is illustrated in Fig. 1, which shows how VM2 obtains a larger share of the capacity over time by creating multiple TCP connections.

Efforts are underway to address this problem by harmonizing the traffic coming from senders directly at the source; this approach has been found to have advantages in terms of scalability and resilience to churn over using switch and router mechanisms such as CoS tags, Weighted Fair Queuing

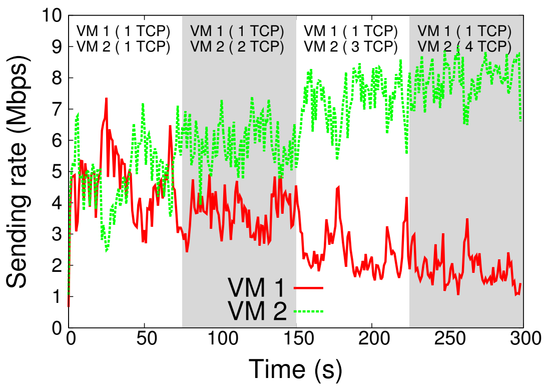


Fig. 1: sending rate of two VMs, with 1 flow in VM1 and 1 to 4 flows in VM2. VM2 aggressively obtains available capacity by opening a new TCP flow every 75 seconds.

or reservations [9]. Bandwidth allocation schemes in general (e.g., EyeQ [3], Gatekeeper [10], Oktopus [5], Secondnet [6], Netshare [11], and FairCloud [7]) tend to operate on a VM-level, making them insufficient to relieve the network of congestion [12].

Mechanisms such as Seawall [9], VCC [13] and AC/DC [12] successfully achieve this sender-side control by running dedicated congestion control algorithms as part of the hypervisor infrastructure. However, they all face a common difficulty, which we address in this paper: how should the new algorithm that is running as part of the hypervisor communicate with the the guest OS?

VCC and AC/DC do not require updating the guest OS at all, which is a significant advantage: it does not require cooperation of tenants to update the OS (if they do bring their own OS), which reduces burden and allows to *enforce* cooperative behavior. However, these approaches also have disadvantages: they have to resort to changing the receive window (rwnd) as a means to control TCP's behavior.[1] A sender can therefore only increase the sending rate as quickly as the TCP implementation inside the guest OS allows. A hypervisor could speed up the TCP sender inside the guest OS by splitting the TCP connection to shorten the control loop, and sending ACKs faster than the real receiver; this requires managing an additional buffer inside the hypervisor, making

[1]Many of the alternatives discussed in [13] have similar limitations: buffering packets or ACKs, duplicating ACKs, splitting connections, etc. The only viable alternative listed is to directly access the guest memory, albeit with some disadvantages as also discussed in [13].

the solution significantly more complex than the approach that we present in this paper.

Seawall takes a different approach: in the guest OS, congestion control implementations need to defer all congestion control decisions to the hypervisor by always asking for allowance before sending a packet [9] (similar to the Congestion Manager (CM) [14]). Seawall alone takes care of congestion control. According to [9], the sheer performance gain should provide enough incentive for tenants to upgrade their OS; this is confirmed by some of our findings (e.g. the significantly shorter completion times of short flows in Fig. 4). However, Seawall needs more drastic changes to the infrastructure than e.g. VCC and AC/DC: both the sender and receiver side are altered, and bits from the header are re-purposed to implement the necessary signaling.

Is there a middle ground? Can we find a way to change the guest OS that is simple and generic, allowing to control TCP with a sender-side only algorithm? What is the missing piece here, between requiring significant guest OS updates and implementing a novel control loop between hypervisors on one hand, and changing nothing but being left with only the receive window to control on the other?

We argue that a middle ground can be found when keeping the guest OS congestion control intact, yet allowing a controlling entity to overrule its decisions. Making use of existing congestion control code is however close to impossible with the "ask to send" interface of the Congestion Manager:[2] because the CM does not have knowledge about the congestion control mechanisms it is talking to, it is limited to either carrying out relatively simple scheduling decisions or implementing a complete congestion control mechanism by itself (which is what it really does).

Our novel contribution in this paper is two-fold: 1) we present a new interface to communicate between TCP in the guest OS and a hypervisor. We call a set of TCP connections that are controlled via this interface *controlled TCP (ctrlTCP)*. 2) Using both ns-2 and FreeBSD implementations, we present the efficacy of our solution by extending our previously proposed sender-side-only *ctrlTCP* algorithm [15]. This allows precise control over intra-flow bandwidth sharing in the multi-tenant data centers, and yields several more advantages of coupling congestion controls.

The rest of our paper is organized as follows: After a review of prior works in Section II, we introduce our *ctrlTCP* design for datacenters in Section III. In Section IV, we present some simulation and experimental results, and Section V concludes the paper.

II. Prior Work

Here we provide an overview of the most relevant related research works, which we categorize according to the scopes

[2]The interface in [14] contains a `cm_update call`, which conveys information such as "type of loss" and "round-trip time (RTT)". While this call is better aligned with our proposal, it also does not fit the bill: the conveyed information is *input* to a congestion control mechanism — but leveraging existing congestion control code requires to convey the *output* of a congestion control mechanism instead.

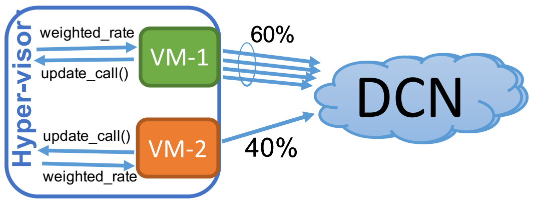


Fig. 2: Abstract *ctrlTCP* architecture, showing an example where the hypervisor removes selfish VM behavior.

of the mechanisms.

Datacenter capacity management: There are several efforts to fairly control and share the network capacity of the complete datacenter. In EyeQ [3] bandwidth is shared and guaranteed across all users of the datacenter and access is controlled at the edges. To achieve fairness, FairCloud [7] uses per-flow queues in the switches and HyGenICC [4] presents a network abstraction layer to each VM. In order to ensure that all VMs get their fair share, somewhat static allocation of bandwidth is performed by Oktopus [5] (coordinated centrally), SecondNet [6] (between pairs of VMs) and Gatekeeper [16].

Generally, most schemes to manage datacenter traffic operate on the *data channel*. This has the advantage that on-host mechanisms for example Seawall and EyeQ can control *all* traffic leaving the VM, not only TCP, and apply functions ranging from congestion control to traffic shaping or scheduling. However, it is not uncommon for hypervisors to enable direct access to hardware drivers (e.g. VMWare's ESXi hypervisor supports TCP Segment Offloading (TSO) for VMs), and this puts such traffic management functions on a critical path regarding execution time. Our scheme is able to utilize such direct access to hardware drivers because, as Fig. 2 shows, *ctrlTCP* operates strictly on the *control path*: it communicates signals (state variables) between the congestion control implementations running in the guest OS and the hypervisor but does not need to even examine or count the number of outgoing or incoming data packets.

Single-path congestion control coupling: Previous work on TCP state sharing has recognized that much can be achieved in a simpler fashion, by sharing a number of state variables [17, 18]. These proposals are closer in spirit to ours, however, they really *only* share variables instead of defining an interface to an algorithm. As we will see, an algorithm is needed to take care of states of TCP. Our own prior work is more recent [19]; it also considers sharing congestion control state across multiple flows, but for multimedia applications in the context of WebRTC. Different from [17, 18], an interface to the congestion management system is proposed in [19], which we have adapted for TCP in [15].

While these prior works already show that our method does a good job at controlling multiple congestion control instances, datacenters have not been the focus of any of these previously published papers. Here, for the first time, we consider the API that TCP instances must use, and implemented the coupling entity in a hypervisor to control flows in the guest OS.

Multiplexing: Another method to avoid the competition of multiple flows is to merge application-layer data streams above a single transport instead of changing the congestion control mechanism to work together. This is done by HTTP/2 [20], for example, which multiplexes web sessions on top of a single TCP connection between client and server. Multiplexing application flows onto a single TCP connection can result in a head-of-line (HoL) blocking, where faster application-layer threads are forced to wait while serialized messages from slower threads are handled at the TCP destination. Solving HoL blocking usually involves entirely different transport protocols, such as QUIC [21] or SCTP [22].

Multi-path congestion control coupling: Coupled congestion control is an important part of MultiPath TCP (MPTCP) [23]. There are several proposals, e.g. LIA [24, 25], OLIA [26] and BALIA [27]. Similar to E-TCP and the CM, these mechanisms try to make multiple flows behave like a single flow when they traverse a single bottleneck (and [28] proposes to detect whether shared bottlenecks exist and switch behavior accordingly). However, MPTCP's coupling assumes that flows could take a different path, and ideally also traverse different bottlenecks.

MPTCP's subflows also use different tuples in order to be able to use different paths. On the contrary, in our mechanism, shared bottlenecks are *assumed*, rendering it unsuitable when TCP connections between the same source and destination pair are placed on different paths. Which mechanism to use under which circumstances is a policy decision, depending on how the administrator of the datacenter wishes to control fairness and load-balance traffic.

III. Our Solution

We present a new interface to communicate between TCP in the guest OS and a hypervisor. We call a set of TCP connections that are controlled via this interface *ctrlTCP_int*. We extended our *ctrlTCP* algorithm from [15] that emulates the behavior of a single TCP congestion controller (much like the CM) and supports priorities (for practical management of both inter- and intra-VM capacity allocation).

Each TCP session communicates with an entity that we call a Coupled Congestion Controller (CCC). As the name suggests, a CCC typically makes decisions that combine the collected knowledge that it receives from all TCP instances that talk to it (thereby "coupling" them in some way). A CCC can operate in a hypervisor (as shown in Fig. 2, and done for the test shown in Fig. 3) or in an OS.

Our algorithm shares state variables across multiple TCP Connections. There are three phases: i) *Register:* TCP connections register with the CC upon joining, providing a connection identifier `c`, a priority `p`, their congestion window `cwnd`, and their slow start threshold `ssthresh` ii) *Update:* whenever a connection changes its cwnd, it updates its status with the CCC. CCC then responds with the calculated cwnd and ssthresh values. iii) *Leave:* when a connection terminates, CCC removes its variables and states and recalculates the aggregates.

Using the *ctrlTCP_int* interface, it is possible to develop a variety of algorithms that combine the individual congestion controls in some way. For example, LISA [29] is a simple algorithm that shares the cwnd of MPTCP subflows in slow start. This happens only at the moment when new subflows join (while a range of coupled congestion control mechanisms exist for MPTCP, they only work in congestion avoidance, leading to bursts in slow start as multiple subflows start up at the same time, potentially across the same bottleneck). With *ctrlTCP_int*, the LISA algorithm would only share the cwnd in the "Register" phase, provided that an existing flow in the same coupled group is in slow start. The "Update" phase would only be used to note which flows have left slow start, and the "Leave" phase would remove flows from the list of flows the CCC keeps track off.

Implementing the Congestion Manager [14] with *ctrlTCP_int* is also straightforward: the input from parameters in the "register" and "update" calls could generally be ignored, and the "accept" and "response" calls would be used to dictate the cwnd that a flow should use. Implementing "ask to send" would just mean that the internally-maintained CCC cwnd value would increase or decrease as determined by the CCC logic and then be given to TCP such that it can either send more data or not. This makes an internal variable explicitly visible to the outside, but changes nothing else.

To summarize, the required changes to TCP are:

- This function call, to be executed at the beginning of a TCP session:
  ```
  register(c, p, cwnd, sshtresh);
  returns: cwnd, ssthresh, state
  ```
- This function call, to be executed whenever TCP newly calculates cwnd:
  ```
  update(c, cwnd, sshthresh, state);
  returns: cwnd, ssthresh, state
  ```
- This function call, to be executed whenever a TCP session ends:
  ```
  leave(c)
  ```

IV. results

We have implemented our mechanism in the FreeBSD 11 kernel[3] with state shared across the freely available VirtualBox[4] hypervisor. Figure 3 was produced with this implementation, showing Jain's Fairness Index [30] ($(\sum_{i=1}^{N} x_i(t))^2 / N \sum_{i=1}^{N} x_i(t)^2$) for $N = 2$ aggregate flows x_1 and x_2, calculated using the traffic that originated from two VMs across a 10 Mbit/s$\times$100 ms bottleneck with and without the coupled congestion control algorithm. It can be seen from Fig. 3 that fairness between two VMs, with 1 flow in VM1 and 1 to 4 flows in VM2. Without coupling, fairness deteriorates; *ctrlTCP* algorithm achieves perfect fairness.

We complement our real-life tests with simulations using ns-2. Simulations have the advantage of not being affected by

[3]the source code is available at: http://safiquli.at.ifi.uio.no/tcp-ccc/

[4]https://www.virtualbox.org

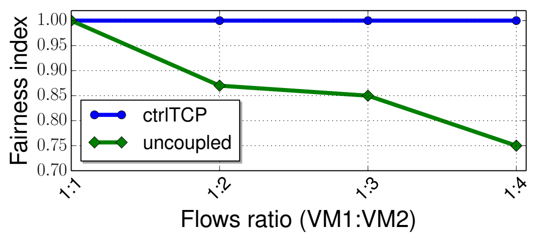


Fig. 3: Fairness between two VMs, with 1 flow in VM1 and 1 to 4 flows in VM2. Without coupling, fairness deteriorates; *ctrlTCP* algorithm achieves perfect fairness.

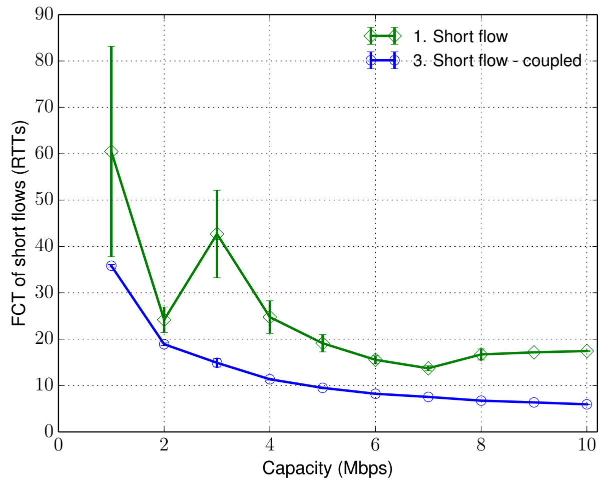
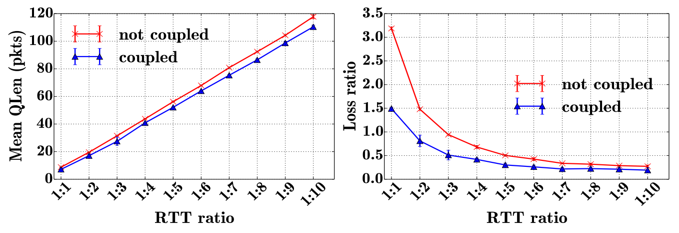


Fig. 4: Flow completion time (FCT) of short flows, with and without *ctrlTCP*



(a) Mean queue length (in pkts)

(b) Loss ratio

Fig. 5: Mean queue length and loss ratio as the RTT ratios between 2 flows is varied

CPU processing power or other OS delay, which can easily become a limit for tests in realistic data center conditions. As Section IV shows, a range of typical datacenter and Internet setups exhibit comparable bandwidth×delay product (BDP) values. The behavior of TCP does not depend on bandwidth or the RTT alone but is a function of the BDP (with the exception of the retransmission timer; however, no timeouts occurred in the simulations that are discussed here). Hence our simulation results apply to both datacenter and Internet scenarios with equal BDP.

To illustrate that our mechanism can significantly reduce short flow completion times (FCT) by giving a large share of the aggregate, we ran a test with a long flow (25 Mb) and a short flow (200 Kb), with and without *ctrlTCP*, while varying the capacity from 1 Mbps to 10 Mbps. The test was repeated 10 times with randomly picked flow start times over the first second for the long flow and the sixth second for the short flow. It can be seen from Fig. 4 that *ctrlTCP* significantly improves the short flow's FCT. Coupling also removes a synchronization effect: in the uncoupled 2 Mbps scenario, the short flow was even faster than in the 3 Mbps scenario because it was the first to send its initial window (IW) into the queue, which did not have enough space for the IWs of two flows. Since the long flow gets to rapidly increase its cwnd when a short flow terminates, our mechanism also reduced the FCT of a long flow, but the impact was negligible.

Since connections in data centers may have different RTTs [32], we therefore consider the case where two connections are originated from the same host, traverse a common bottleneck and arrive at two different destinations. We varied RTT ratios between two flows. Fig. 5(a) and 5(b) illustrate that coupling with our mechanism significantly reduces the average queue length and loss ratio of the connections. Coupling resulted in a small reduction in throughput, but never more than 3%. In these simulations, we added preprocessed TMIX background traffic, taken from a 60-minute trace of campus traffic at the University of North Carolina [33], to get an approximate load of 50% on a 10 Mbps link, with RTTs of the background TCP flows in the range of 80-100 ms.

V. Conclusions

TCP connections between the same endpoints often traverse the same bottleneck and compete with each other in both data centers and the Internet. We have introduced a new interface to communicate between a TCP implementation in a guest OS and a hypervisor such that TCP connections from multiple VMs can be controlled together. This allows datacenter administrators to exert precise control over the relative bandwidth share offered to coupled flows, with only minimal interfacing to the kernel TCP code. Using both ns-2 and FreeBSD implementations, we have explored the benefits of our mechanism.

Adapting our CCC algorithm to be more or less aggressive is straightforward, e.g. by changing the increase/decrease behavior as a function of the number of flows in a coupled group similar to the way in which EFCM [18] differs from E-TCP [17], or the way in which MulTCP [34] differs from TCP. We plan to develop such extensions of our algorithm in future work, and also investigate controlling a larger variety of congestion control mechanisms with it. We also plan to investigate our solution on 10 Gbps links while considering typical practical challenges at high speeds such as CPU delay.

References

[1] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data Center TCP (DCTCP)," in *Proc. of the ACM SIGCOMM*, 2010.

Datacenter	Internet	BDP (1500 byte packets)
10 Gbit/s, 100 μs: [1], fig.12	10 Mbit/s, 100 ms	83.3
10 Gbit/s, 10..100 μs: [2], sec. 3.1	1..10 Mbit/s, 100 ms	8.3 .. 83.3
1 Gbit/s, 100 μs: [4], sec. 5A	10 Mbit/s, 10 ms	8.3
1 Gbit/s, 250 μs: [31], sec. 4.4	25 Mbit/s, 100 ms	208.3

TABLE I: Applicability of simulation results: referenced datacenter conditions are comparable to common Internet bandwidth×delay products. Note that this comparison is only valid for simulations that ignore typical practical challenges at high speeds (CPU delay etc.) and operate without TCP retransmission timeouts (RTOs).

[2] R. Mittal, V. T. Lam, N. Dukkipati, E. Blem, H. Wassel, M. Ghobadi, A. Vahdat, Y. Wang, D. Wetherall, and D. Zats, “Timely: Rtt-based congestion control for the datacenter,” in *Proc. of the ACM SIGCOMM*, 2015.

[3] V. Jeyakumar, M. Alizadeh, D. Mazières, B. Prabhakar, A. Greenberg, and C. Kim, “EyeQ: practical network performance isolation at the edge,” in *Proc of the NSDI’13*, 2013.

[4] A. M. Abdelmoniem, B. Bensaou, and A. J. Abu, “HyGenICC: Hypervisor-based generic IP congestion control for virtualized data centers,” in *proc of the IEEE ICC*, May 2016.

[5] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron, “Towards predictable datacenter networks,” in *Proc.of the ACM SIGCOMM 2011*.

[6] C. Guo, G. Lu, H. J. Wang, S. Yang, C. Kong, P. Sun, W. Wu, and Y. Zhang, “SecondNet: A data center network virtualization architecture with bandwidth guarantees,” in *Proceedings of the Co-Next’10*, 2010.

[7] L. Popa, G. Kumar, M. Chowdhury, A. Krishnamurthy, S. Ratnasamy, and I. Stoica, “FairCloud: Sharing the network in cloud computing,” in *Proceedings of the ACM SIGCOMM 2012*.

[8] A. Afanasyev, N. Tilley, P. Reiher, and L. Kleinrock, “Host-to-host congestion control for TCP,” *Commun. Surveys Tuts.*, vol. 12, no. 3, pp. 304–342, Jul. 2010.

[9] A. Shieh, S. Kandula, A. Greenberg, C. Kim, and B. Saha, “Sharing the data center network,” in *Proc. of NSDI*, 2011.

[10] H. Rodrigues, J. R. Santos, Y. Turner, P. Soares, and D. Guedes, “Gatekeeper: Supporting bandwidth guarantees for multi-tenant datacenter networks.” in *WIOV*, 2011.

[11] V. T. Lam, S. Radhakrishnan, R. Pan, A. Vahdat, and G. Varghese, “Netshare and stochastic Netshare: Predictable bandwidth allocation for data centers,” *SIGCOMM Comput. Commun. Rev.*, vol. 42, no. 3, pp. 5–11, Jun. 2012.

[12] K. He, E. Rozner, K. Agarwal, Y. J. Gu, W. Felter, J. Carter, and A. Akella, “AC/DC TCP: Virtual congestion control enforcement for datacenter networks,” in *Proc. of SIGCOMM*, 2016.

[13] B. Cronkite-Ratcliff, A. Bergman, S. Vargaftik, M. Ravi, N. McKeown, I. Abraham, and I. Keslassy, “Virtualized congestion control,” in *Proc. of the ACM SIGCOMM*, 2016.

[14] H. Balakrishnan, H. S. Rahul, and S. Seshan, “An integrated congestion management architecture for internet hosts,” in *Proc. of the ACM SIGCOMM*, 1999.

[15] S. Islam, M. Welzl, H. Kristian, D. Hayes, G. Armitage, and S. Gjessing, “ctrlTCP, reducing latency through coupled heterogeneous multi-flow TCP congestion control,” in *Proc of the IEEE GI*, 2018.

[16] H. Rodrigues, J. R. Santos, Y. Turner, P. Soares, and D. Guedes, “Gatekeeper: Supporting bandwidth guarantees for multi-tenant datacenter networks,” in *Proceedings of the WIOV’11*, 2011.

[17] L. Eggert, J. Heidemann, and J. Touch, “Effects of ensemble TCP,” *USC/ISI*, vol. 7, no. 1, 1999.

[18] 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*, vol. 49, no. 2, pp. 269–294, 2005.

[19] S. Islam, M. Welzl, S. Gjessing, and N. Khademi, “Coupled congestion control for RTP media,” in *SIGCOMM Computer Communication Rev.*, vol. 44, no. 4. ACM, 2014, pp. 21–26.

[20] M. Belshe, R. Peon, and M. Thomson, “Hypertext Transfer Protocol Version 2 (HTTP/2),” RFC 7540 (Proposed Standard), Internet Engineering Task Force, May 2015.

[21] A. Langley, A. Riddoch, A. Wilk, A. Vicente, C. B. Krasic, C. Shi, D. Zhang, F. Yang, F. Kouranov, I. Swett, J. Iyengar, J. Bailey, J. C. Dorfman, J. Roskind, J. Kulik, P. G. Westin, R. Tenneti, R. Shade, R. Hamilton, V. Vasiliev, and W.-T. Chang, “The quic transport protocol: Design and internet-scale deployment,” 2017.

[22] L. Ong and J. Yoakum, “An Introduction to the Stream Control Transmission Protocol (SCTP),” RFC 3286 (Informational), Internet Engineering Task Force, May 2002.

[23] C. Raiciu, C. Paasch, S. Barre, A. Ford, M. Honda, F. Duchene, O. Bonaventure, and M. Handley, “How hard can it be? designing and implementing a deployable multipath TCP,” in *Proc. of NSDI*, 2012.

[24] H. Li, R. Zheng, and A. Farrel, “Multi-Segment Pseudowires in Passive Optical Networks,” RFC 6456 (Informational), Internet Engineering Task Force, Nov. 2011.

[25] D. Wischik, C. Raiciu, A. Greenhalgh, and M. Handley, “Design, implementation and evaluation of congestion control for multipath TCP,” in *Proc. of the NSDI’11*, 2011, pp. 99–112.

[26] R. Khalili, N. Gast, M. Popovic, U. Upadhyay, and J.-Y. Le Boudec, “MPTCP is not pareto-optimal: Performance issues and a possible solution,” in *Proc. of the CoNEXT*, 2012.

[27] Q. Peng, A. Walid, J. Hwang, and S. H. Low, “Multipath TCP: Analysis, design, and implementation,” *IEEE/ACM Transactions on Networking*, vol. 24, no. 1, pp. 596–609, Feb 2016.

[28] S. Ferlin, O. Alay, T. Dreibholz, D. A. Hayes, and M. Welzl, “Revisiting congestion control for multipath TCP with shared bottleneck detection,” in *Proc. of the IEEE INFOCOM*, 2016.

[29] R. Barik, M. Welzl, S. Ferlin, and O. Alay, “LISA: A linked slow-start algorithm for MPTCP,” in *Proc of the (ICC)*, 2016.

[30] R. Jain, D. Chiu, and W. Hawe, “A Quantitative Measure of Fairness and Discrimination for Resource Allocation in Shared Computer Systems,” DEC Research, Tech. Rep. TR-301, 1984.

[31] A. Munir, G. Baig, S. M. Irteza, I. A. Qazi, A. X. Liu, and F. R. Dogar, “Friends, not foes: Synthesizing existing transport strategies for data center networks,” in *Proc. of the ACM SIGCOMM*, 2014.

[32] B. Vamanan, J. Hasan, and T. Vijaykumar, “Deadline-aware datacenter TCP (D2TCP),” in *Proc. of the ACM SIGCOMM*. New York, NY, USA: ACM, 2012, pp. 115–126.

[33] M. C. Weigle, P. Adurthi, F. Hernández-Campos, K. Jeffay, and F. D. Smith, “Tmix: A tool for generating realistic TCP application workloads in ns-2,” *SIGCOMM Comput. Commun. Rev.*, vol. 36, no. 3, pp. 65–76, Jul. 2006.

[34] J. Crowcroft and P. Oechslin, “Differentiated end-to-end internet services using a weighted proportional fair sharing TCP,” *SIGCOMM CCR*, vol. 28, no. 3, pp. 53–69, 1998.