ctrlTCP: Reducing Latency through Coupled, Heterogeneous Multi-Flow TCP Congestion Control

Safiqul Islam*, Michael Welzl*, Kristian Hiorth*, David Hayes†, Grenville Armitage‡, Stein Gjessing*

*University of Oslo, Norway
†Simula Research Laboratory, Norway
‡Swinburne University of Technology, Australia

***Abstract*—We present *ctrlTCP*, a method to combine the congestion controls of multiple TCP connections. In contrast to the previous methods such as the Congestion Manager, *ctrlTCP* can couple *all* TCP flows that leave one sender, traverse a common bottleneck (e.g., a home user's thin uplink) and arrive at different destinations. Using ns-2 simulations and an implementation in the FreeBSD kernel, we show that our mechanism reduces queuing delay, packet loss, and short flow completion times while enabling precise allocation of the share of the available bandwidth between the connections according to the needs of the applications.**

I. Introduction

Multiple TCP connections are often initiated from the same host. They can overlap in time and may share the same bottleneck. When this happens, they can sub-optimally interact (and indeed, compete) with each other and other flows. These overlapping connections that are intended to provide throughput gains will also often push up queuing delays and/or packet loss rates, to the detriment of flows sharing any bottlenecks along the common path. In large part this is due to each overlapping TCP connection's congestion control state machines acting independently.

Significant solutions to date fall into one of two classes: Merge common application-layer data streams onto a single transport layer connection, or couple the transport layer congestion control machinery for connections known to share the same endpoints. Examples of the former include SPDY [1] and HTTP/2 [2], which multiplex multiple web sessions on top of a single TCP connection between client and server. Examples of the latter, often referred to as *coupled congestion control (ccc)*, include the Congestion Manager (CM [3]), Ensemble TCP (E-TCP [4]) and Ensemble Flow Congestion Management (EFCM [5]).

However, there are known problems within each class of solutions. For instance, simply multiplexing application flows onto a single TCP connection may result in 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. Packet loss on one flow delays all multiplexed flows until the loss can be recovered. Solving HoL blocking usually involves entirely different transport protocols, such as QUIC [6] or SCTP [7].

On the other hand, all the coupled congestion control strategies are designed for connections between the same endpoints having homogeneous round-trip times, and they tend not to fully leverage the statefulness of the TCP congestion control algorithm. Their mechanisms rely on the assumption that connections between the same endpoints share a common bottleneck — this may be true for routers in theory, but in reality two connections may take different routes if they have different TCP port numbers [8].

A coupled congestion control mechanism only makes sense when flows traverse a shared bottleneck. Identifying a shared bottleneck is therefore very important. To the best of our knowledge, there are three different approaches to derive a shared bottleneck: i) via *multiplexing* (which is a completely reliable method where flows are multiplexed onto a single transport layer connection, e.g., VPNs, WebRTC), ii) via *configuration* (e.g., a home user's thin uplink, a common wireless bottleneck or by Software Defined Networking), and iii) via *measurements* (e.g., correlations among measured delay and losses can be used to deduce a common bottleneck).

Whenever the first two approaches are available, we can readily apply a ccc approach. A configuration based approach requires prior knowledge about the network environment. If known, a ccc mechanism can combine flows (having heterogeneous RTTs) originated from the same host that are destined for different receivers, otherwise a measurement based shared bottleneck detection mechanism [9, 10] should be used.

We introduce *ctrlTCP*, a refined coupled congestion control strategy that better utilizes a TCP sender's awareness of network conditions and allows precise bandwidth sharing. The novel contribution of our mechanism is that it 1) can combine *any* flows that share a bottleneck—this gives us the flexibility to couple TCP flows that leave one sender, traverse a common bottleneck (e.g., a home user's thin uplink) and arrive at different destinations— and 2) works better than prior work: it is less intrusive and easier to implement than the CM, easier to dynamically enable / disable than multiplexing / scheduling based methods, and does not exhibit problems that we find with E-TCP and EFCM (see Section III).

Using both ns-2 and FreeBSD implementations we have explored the benefits of our coupled congestion control scheme. Our results demonstrate significantly better (lower) queuing delays, packet loss rates, and short flow completion times compared to uncoupled TCP connections without significantly affecting throughput or long flow completion times. We believe

978-1-5386-5541-2/18/$31.00 ©2018 IEEE
DOI 10.1109/INFCOMW.2018.8406887

our approach is practical and deployable in today's Internet, offering lower RTTs to all traffic sharing bottlenecks with coupled TCP connections.

The rest of our paper is organized as follows: Section II introduces related work and Section III describes our improved approach to coupled TCP congestion control. We present simulations and experimental results in Section IV. The paper concludes in Section V.

II. Related Work

To the best of our knowledge, RFC 2140 [11] was the first work to outline a mechanism for coupling TCP connections by sharing the TCP Control Block (TCB) in order to better initialize new connections. This idea was expanded by Ensemble TCP (E-TCP) [4] to allow concurrent flows to benefit from each other beyond initialization, working together so that the aggregate is no more aggressive than a single TCP flow. On the other hand, Ensemble Flow Congestion Management (EFCM) [5] allows the aggregate to be collectively as aggressive as the combination of separately controlled TCP connections. The Congestion Manager (CM) [3, 12] takes the concept even further, completely replacing each flow's congestion controller with its own congestion control mechanism—a rate based controller.

The CM is a major modification to the implementation of congestion control as part of TCP. Our proposal aims to minimize changes to the kernel TCP code, making it much easier to implement as an add-on, closer in spirit to E-TCP and EFCM, but fixing problems that have identified with these mechanisms.

Although having important differences, congestion control coupling (as described here) shares some similarities when compared to coupled congestion control for MultiPath TCP (MPTCP), e.g. the mechanisms LIA [13], OLIA [14] and BALIA [15]. Similar to our proposal, E-TCP and the CM, these mechanisms try to behave like one flow through a shared resource. However, MPTCP's coupling assumes that flows take different paths, and therefore are likely to also traverse different bottlenecks. This means that some things that *ctrlTCP* or other coupled congestion control mechanisms can do would probably be quite inappropriate for MPTCP. For example with *ctrlTCP*:

- A new connection joining the aggregate can immediately get a share of the potentially quite large cwnd of ongoing transfers. This can significantly reduce the completion time of short flows, but in MPTCP this would be inappropriate since it results in using a very large initial window on a new path.
- Changing states (e.g. avoiding slow start), as our algorithm does, strongly relies on the knowledge that there is only a single shared bottleneck. The resulting behaviour would be quite wrong in case of multiple bottlenecks.
- A share of the aggregate $cwnd$ can be assigned based on application preference in the case of *ctrlTCP*. These preferences can even change on the fly. Again, if flows were to traverse different paths, the send rate on a particular path could temporarily be much more aggressive than TCP.

Finally, MPTCP's subflows also use different identifying tuples *in order to* be able to use different paths – this is, for example, leveraged in [16] without even using multi-homed end-systems.

III. *ctrlTCP* Algorithm Design

Despite a number of previous attempts at coupled congestion control, widespread deployment has proven to be difficult, mainly due to the aforementioned complexities of the proposed mechanisms. We draw inspiration from a method used to couple congestion control for media flows in WebRTC [17, 18]. TCP has particular difficulties due to its stateful nature that require a significantly different design approach for *ctrlTCP*. We first describe the key elements of the basic algorithm logic in Section III-A, and then illustrate in Section III-B how *ctrlTCP* solves key problems inherent in the E-TCP and EFCM mechanisms, and finally in Section III-C we introduce a novel approach to avoid sudden bursts in the network that "paces" the packets by simply maintaining the ACK-clocks of the TCP connections.

A. Basic algorithm logic

In *ctrlTCP* each TCP session communicates with an entity called a Coupled Congestion Controller (CCC). The CCC couples flows traversing the same bottleneck and stores some shared variables including sum_cwnd and $sum\_ssthresh$.

New TCP sessions first register with the CCC, supplying it with a (i) Priority (P), (ii) cwnd, and (iii) ssthresh. The newly joining flow then obtains its first cwnd and ssthresh values.

When the TCP state of a coupled flow is changed, the flow sends an update message to the CCC including cwnd, sshthresh, RTT and the TCP state machine state. The CCC responds by assigning calculated values of cwnd and ssthresh to the updating flow. We emulate the behavior of a single TCP session by choosing the flow with the shortest RTT as the Coordinating Connection (CoCo), since it has the most timely congestion information. This flow dictates the increase behavior for the aggregate while in the TCP Congestion Avoidance phase. For simplicity, the algorithm refrains from adjusting cwnd when a connection is in Fast Recovery (FR). As we will explain in Section III-B, we limit the usage of Slow Start (SS), ensuring that the aggregate's behavior is only dictated by SS when all connections are in the SS phase.

When a TCP flow terminates its variables are removed and the summations are recalculated. The aggregate variables also need to be adjusted when a flow leaves. When the last flow in the coupled group terminates, the group is removed from the CCC.

ctrlTCP shares state variables across multiple TCP connections. This has been done in the past by E-TCP [4] and EFCM [5], however, these works do not properly account for the stateful nature of TCP. This can lead to erroneous behavior if the goal is to emulate the dynamics of a single

TCP connection. We now use E-TCP and EFCM to illustrate the merits of the design rationale underlying our algorithm.

B. Handling Loss events

1) Fast recovery behaviour: ctrlTCP observes when a flow enters Fast Recovery (FR) and reacts by reducing $sum\_ssthresh$ and sum_cwnd appropriately. As long as a flow is in FR, we refrain from overriding that flow's $cwnd$ and $ssthresh$ values to allow the Fast Recovery/Fast Retransmit logic to work.

TCP operates on loss events (one or more packet losses per RTT), not individual packet losses. When congestion controls are combined, this logic should be preserved. We explain this by using an example of two connections traversing the same bottleneck. A single packet drop from connection 1, two drops from connections 1 and 2 or multiple packet drops from connection 2 only, should all result in the same behavior of the traffic aggregate. Simply sharing TCP variables such as $cwnd$ or $ssthresh$ cannot achieve this.

In order to compare our mechanism with EFCM coupling[1] and without coupling, we simulated the behaviour of two TCP Reno connections in the ns-2 simulator[2] using a dumbbell topology (bottleneck capacity 10 Mbps, RTT 100 ms, packet size 1500 bytes, and queue length of one BDP (83 packets)). Fig. 1(a–d) captures the behavior of two TCP Reno flows, without coupling, with EFCM, and with our proposed coupled congestion control mechanism, respectively. We artificially induced a packet drop for connection 1 at 25 seconds. Fig. 1(a) shows that without coupling, it takes quite some time for the two flows to converge, and with EFCM (see Fig. 1(b)), the aggregate is not halved, and the flows remain aggressive. This can lead to higher queue growth and packet losses.

2) Timeouts and Slow Start: ctrlTCP operates on the principle that slow start following a timeout should only happen if *no* packets could be delivered across the same path for the timeout interval. This assumption is broken if some but not all connections experience a timeout. In practice this means that if at least one of the flows is still receiving acknowledgements and has not tried to enter slow start, then it is not appropriate for the coupled group to enter slow start.

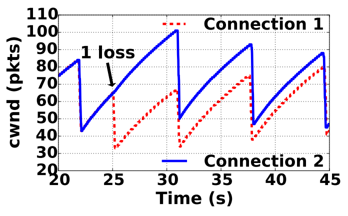
E-TCP does not adopt this rationale, instead forcing all flows to slow start if any one has a timeout (see Fig. 1(c) where we artificially induced a timeout for connection 1 to simulate this behavior). EFCM has the issue of sharing the initial very large $ssthresh$ value, which potentially moves all existing flows back into Slow Start from Congestion Avoidance when a new flow joins.

With *ctrlTCP*, if the timeout of a particular flow is indeed due to lost packets, it will recover the lost packets when it enters Fast Recovery. In the extreme case when a flow has lost its entire cwnd of packets, it can take up to 1+Dupthresh[3]

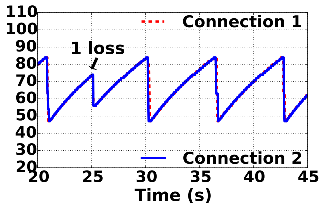
[1]We implemented EFCM and E-TCP in ns-2 based on the descriptions in [5] and [4], respectively.

[2]We used the TCP-Linux module that gives the flexibility to use the actual Linux TCP code in simulations. In our case, it was Linux kernel 3.17.4

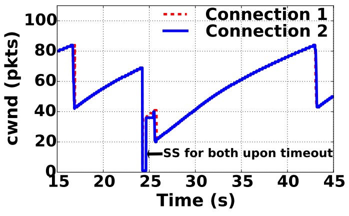
[3]Number of duplicate ACKs required to trigger fast recovery



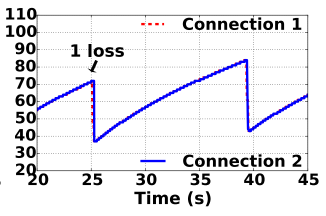
(a) Not coupled



(b) Coupled with EFCM



(c) Coupled with E-TCP



(d) Coupled with *ctrlTCP*

Fig. 1: cwnd of two TCP Reno flows

RTTs for the flow to enter fast recovery. The reason for this is that the flow's duplicate ACK counter is not being incremented for ACKs received on other coupled flows, as would be the case if it was truly one aggregated TCP flow. A possible solution to this — currently under investigation — is to count ACKs from other coupled flows and send this DupACK count when the affected flow sends its update. This will allow fast recovery to be triggered after just 1 RTT and still maintains the simplicity of implementation and interaction with the coupled flows that our algorithm has.

C. ACK-clocking

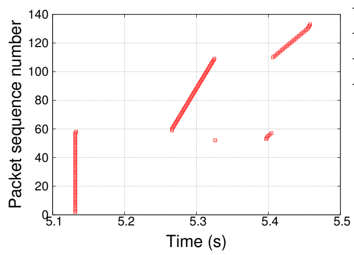
When a new connection joins, it may benefit from sharing the large cwnd of existing flows. This has potential to cause bursts of transmission into the network, unless the packets are paced in some way. Consider the scenario where connection 1 has already achieved a cwnd of 100 packets, connection 2 joins and receives cwnd=50 packets as its share of the capacity. If it sends these without some form of pacing it can cause a significant congestion spike in the network. It is not a problem for connection 1 alone because its packets are paced by arriving ACKs.

ctrlTCP uses a simple ACK-clocking mechanism to avoid these bursts. Rather than using timers, in this example we utilize the acknowledgements connection 1 receives to pace the sending of connection 2 over the course of the first RTT. In this way, we avoid causing a congestion spike in the network.

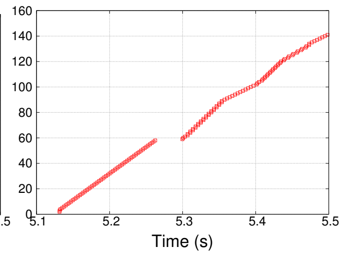
Fig. 2 shows the packet sequence diagrams over time of two coupled-TCP Reno connections, with and without ACK-clocking. Without ACK-clocking, the congestion spike causes significant packet loss. Our ACK-clocking algorithm completely eliminates this issue.

IV. Results

We have implemented *ctrlTCP* in the ns-2 simulator and in the FreeBSD-11 kernel. In the simulations, we used pre-



(a) Packet sequence diagram of connection 2, without ACK-clocking



(b) Packet sequence diagram of connection 2, with ACK-clocking

Fig. 2: Packet sequence plots of 2 flows when flow 2 joins after 5 seconds.

processed TMIX background traffic[4] in order to provide an approximate load of 50% on a 10 Mbps bottleneck link. The RTTs of background TCP flows generated by TMIX are in the range of 80–100 ms. For the emulation experiments the sender and receiver machines are physical, identical desktop computers (Intel i7-870 2.93GHz CPU, 8GB RAM) equipped with Gigabit Ethernet Network Interface Cards (NICs), running our modified version of FreeBSD-11. They are connected via a third identical machine running Ubuntu Linux 15.04 and the CORE network emulator [21] version 4.7 to form the dumbbell topology (bottleneck capacity 10 Mbps, RTT 100 ms, MTU 1500 bytes, and queue length 1 BDP (83 packets)). The underlying TCP congestion control mechanism is NewReno.

In this paper, we first show the results for connections carrying bulk traffic with homogeneous RTTs of 100 ms. Later we show the efficacy of our solution when the RTTs of the connections are varied.

A. Connections with Homogeneous RTTs

Fig. 3 showcases an example of *ctrlTCP* where two connections are created with priorities 0.75 and 0.25, respectively. Connection 1 starts at t=3 s and connection 2 starts at t=30 s. It can be seen from Fig. 3 that between t=30 s and t=85 s (the duration when two flows coexist), flow 1 gets 3/4 of the aggregate cwnd while flow 2 gets 1/4. We also see that the two flows are coupled, as they increase and decrease their cwnds at the same time. Outside of this interval, flow 1 gets all of the cwnd. In Fig. 3, flow 2 is able to immediately use its share of the aggregate cwnd.

Fig. 4 demonstrates that *ctrlTCP*'s coupling can yield a significant improvement in the short flow's completion time. We repeated this test 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 show the reduction of flow completion times (FCTs) in emulation while varying the bottleneck capacity (6, 8, and 10 Mbps) in Fig. 4. The impact on the long flow was negligible.

[4]The TMIX traffic used in ns-2 simulations is taken from 60-minute trace of campus traffic at the University of North Carolina [19], and it can be accessed from the common TCP evaluation suite [20].

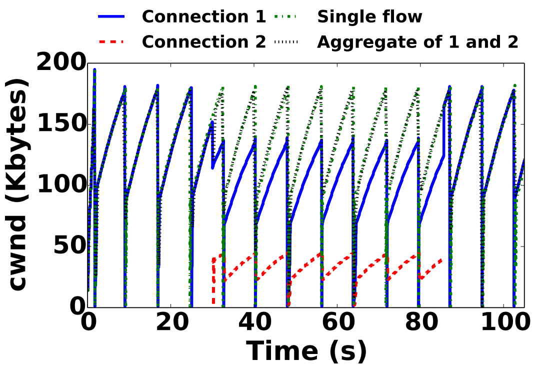


Fig. 3: cwnd (in Kbytes) plot of two TCP connections using coupled congestion control with priorities compared to a single TCP flow scenario. The aggregate line depicts the sum of cwnds in two connections scenario, closely matching a single TCP flow. cwnd going down all the way to 0 whenever cwnd is reduced is not related to *ctrlTCP* but a result of how cwnd is internally updated in FreeBSD.

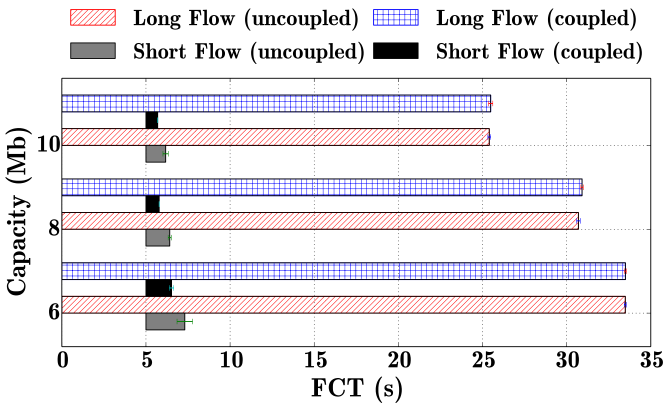
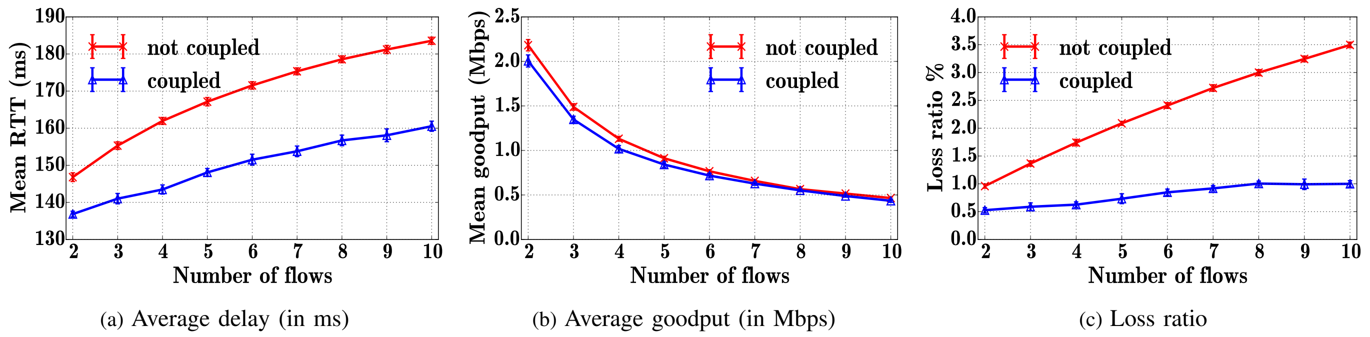


Fig. 4: Flow completion time (FCT) of short flows without ACK-clocking (emulation).

In the following tests, experiments were repeated 10 times with different randomly picked flow start times over the first second. Prerecorded traces of self-similar crosstraffic were injected to occupy 50% of the bottleneck link capacity on average. These were generated using the D-ITG [22] traffic generator, by superposition of 11 on/off streams, with Pareto heavy-tailed on-time distributions ($H = 0.8$) and exponentially distributed off-times ($\mu \in [1, 2]$ s). Packet sizes are normally distributed ($\mu = 1000, \sigma = 200$), with exponentially distributed ($\mu \in [50, 150]$ pps) inter-packet times.

Fig. 5(a) and 5(c) illustrate the average RTT, and loss ratio, with and without coupling for varying numbers of flows, respectively. It can be seen from the graphs that *ctrlTCP* reduces both the average RTT and loss without significantly affecting goodput as we varied the number of flows (see Fig. 5(b)).

Fig. 6 demonstrates that our mechanism can distribute the share of the aggregate cwnd to the TCP connections based on the needs of the applications. Both the simulation and



(a) Average delay (in ms) (b) Average goodput (in Mbps) (c) Loss ratio

Fig. 5: Average delay, average goodput, and loss ratio as the number of TCP connections is varied, with and without coupled congestion control (emulation)

emulation results confirm that our mechanism calculates and assigns the shares almost ideally as we vary the priority ratio between two TCP Reno connections.

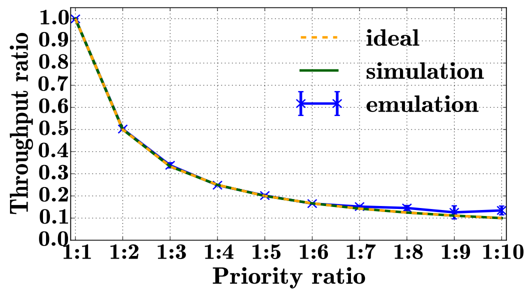


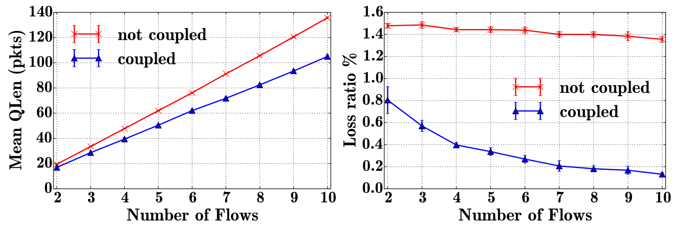
Fig. 6: Throughput ratio as the priorities of two TCP connections are varied

B. Connections with Heterogeneous RTTs

We have also simulated the case where connections are initiated from the same sender, traverse a common bottleneck and arrive at different destinations. We varied the number of flows with different RTT ratios as shown in Table I. Fig. 7(a) and 7(b) illustrate that the average queue length and loss ratio of the connections are consistently lower when coupled with *ctrlTCP*. Coupling resulted in a small reduction in throughput, but never more than 3%.

# of flows	RTT ratio	minRTT	maxRTT	meanRTT
2	1:2	20	40	30
3	1:2:3	20	60	40
4	1....4	20	80	50
5	1....5	20	100	60
6	1....6	20	120	70
7	1....7	20	140	80
8	1....8	20	160	90
9	1....9	20	180	100
10	1:2:3:4:5:6:7:8:9:10	20	200	110

TABLE I: RTT ratio, maxRTT (in ms), minRTT (in ms) and mean RTT (in ms) as the number of flows is varied



(a) Mean queue length (in pkts) (b) Loss ratio

Fig. 7: Mean queue length and loss ratio as the number of flows with different RTT ratios is varied

V. Conclusions

It is increasingly common that a sender initiate several standard TCP connections that overlap in time and hence often traverse a common bottleneck. This may result in competition, rather than cooperation, between each connection's congestion control mechanism, often leading to undesirable spikes in queuing delay and packet loss rates.

We have introduced *ctrlTCP*, a new coupled congestion control strategy that allows applications to exert precise allocation over the relative bandwidth share offered to coupled flows, with only minimal changes to the kernel TCP code.

We have implemented *ctrlTCP* in both ns-2 and the FreeBSD kernel[5], and used these implementations to demonstrate the utility of our proposal: *ctrlTCP* yields lower average queuing delays, lower packet loss rates, and significantly shorter flow completion times for short flows than uncoupled TCP flows while having a negligible impact on overall goodput and long flow completion times. Our solution also avoids several shortcomings we identified in previous mechanisms.

Coupling flows is especially beneficial when many short web-like flows share a common bottleneck as it allows the short flows to quickly obtain a share of the available capacity. There may be scenarios, such as multiple bulk transfers, where coupling flows to competing as a single TCP flow may reduce

[5]The source code of *ctrlTCP* is available at http://safiquli.at.ifi.uio.no/tcp-ccc/

overall performance when compared to multiple single TCP flows. As future work we plan to investigate dynamically tuning our mechanism as suggested by MulTCP [23] and MulTFRC [24] to behave as some appropriate multiple of a TCP flow in certain circumstances, though what is an appropriate multiple in particular circumstances is still contentious. *ctrlTCP* will facilitate experimental investigation of this issue.

VI. Acknowledgments

This work has received funding from 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. The authors would like to thank Øystein Dale for helping with the FreeBSD implementation.

References

[1] "SPDY: An experimental protocol for a faster web," http://www.chromium.org/spdy/spdy-whitepaper.

[2] M. Belshe, R. Peon, and M. Thomson, "Hypertext Transfer Protocol Version 2 (HTTP/2)," RFC 7540 (Proposed Standard), Internet Engineering Task Force, May 2015. [Online]. Available: http://www.ietf.org/rfc/rfc7540.txt

[3] H. Balakrishnan, H. S. Rahul, and S. Seshan, "An integrated congestion management architecture for internet hosts," in *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, ser. SIGCOMM '99. New York, NY, USA: ACM, 1999, pp. 175–187. [Online]. Available: http://doi.acm.org/10.1145/316188.316220

[4] L. Eggert, J. Heidemann, and J. Touch, "Effects of ensemble TCP," *USC/Information Sciences Institute*, vol. 7, no. 1, December 1999.

[5] 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.

[6] J. Iyengar, I. Swett, R. Hamilton, and A. Wilk, "QUIC: A UDP-Based Secure and Reliable Transport for HTTP/2," IETF, I-D draft-tsvwg-quic-protocol-02, Jan. 2016, work in Progress. [Online]. Available: https://tools.ietf.org/html/draft-tsvwg-quic-protocol-02

[7] L. Ong and J. Yoakum, "An Introduction to the Stream Control Transmission Protocol (SCTP)," RFC 3286 (Informational), Internet Engineering Task Force, May 2002. [Online]. Available: http://www.ietf.org/rfc/rfc3286.txt

[8] R. Krishnan, L. Yong, A. Ghanwani, N. So, and B. Khasnabish, "Mechanisms for Optimizing Link Aggregation Group (LAG) and Equal-Cost Multipath (ECMP) Component Link Utilization in Networks," RFC 7424 (Informational), Internet Engineering Task Force, Jan. 2015. [Online]. Available: http://www.ietf.org/rfc/rfc7424.txt

[9] D. Hayes, S. Ferlin, M. Welzl, and K. Hiorth, "Shared Bottleneck Detection for Coupled Congestion Control for RTP Media." Internet Engineering Task Force, Internet-Draft draft-ietf-rmcat-sbd-09, Nov. 2017, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-ietf-rmcat-sbd-09

[10] D. A. Hayes, S. Ferlin, and M. Welzl, "Practical passive shared bottleneck detection using shape summary statistics," in *2014 IEEE 39th Conference on Local Computer Networks (LCN)*. IEEE, 2014, pp. 150–158.

[11] J. Touch, "TCP Control Block Interdependence," RFC 2140 (Informational), Internet Engineering Task Force, Apr. 1997. [Online]. Available: http://www.ietf.org/rfc/rfc2140.txt

[12] H. Balakrishnan and S. Seshan, "The Congestion Manager," RFC 3124 (Proposed Standard), Internet Engineering Task Force, Jun. 2001. [Online]. Available: http://www.ietf.org/rfc/rfc3124.txt

[13] H. Li, R. Zheng, and A. Farrel, "Multi-Segment Pseudowires in Passive Optical Networks," RFC 6456 (Informational), Internet Engineering Task Force, Nov. 2011. [Online]. Available: http://www.ietf.org/rfc/rfc6456.txt

[14] R. Khalili, N. Gast, M. Popovic, and J.-Y. L. Boudec, "Opportunistic Linked-Increases Congestion Control Algorithm for MPTCP," Internet Engineering Task Force, Internet-Draft draft-khalili-mptcp-congestion-control-05, Jan. 2015, work in Progress. [Online]. Available: https://tools.ietf.org/html/draft-khalili-mptcp-congestion-control-05

[15] A. Walid, Q. Peng, S. H. Low, and J. Hwang, "Balanced Linked Adaptation Congestion Control Algorithm for MPTCP," Internet Engineering Task Force, Internet-Draft draft-walid-mptcp-congestion-control-04, Jan. 2016, work in Progress. [Online]. Available: https://tools.ietf.org/html/draft-walid-mptcp-congestion-control-04

[16] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley, "Improving datacenter performance and robustness with multipath TCP," in *Proceedings of the ACM SIGCOMM 2011 Conference*, ser. SIGCOMM '11. New York, NY, USA: ACM, 2011, pp. 266–277. [Online]. Available: http://doi.acm.org/10.1145/2018436.2018467

[17] S. Islam, M. Welzl, and S. Gjessing, "Coupled congestion control for RTP media," Internet Engineering Task Force, Internet-Draft draft-ietf-rmcat-coupled-cc-07, Sep. 2017, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-ietf-rmcat-coupled-cc-07

[18] S. Islam, M. Welzl, S. Gjessing, and N. Khademi, "Coupled congestion control for RTP media," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 4, pp. –, Aug. 2014. [Online]. Available: http://doi.acm.org/10.1145/2630088.2630089

[19] 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. [Online]. Available: http://doi.acm.org/10.1145/1140086.1140094

[20] D. Hayes, D. Ros, L. L. Andrew, and S. Floyd, "Common TCP Evaluation Suite," Internet Engineering Task Force, Internet-Draft draft-irtf-iccrg-tcpeval-01, Jul. 2014, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-irtf-iccrg-tcpeval-01

[21] J. Ahrenholz, C. Danilov, T. R. Henderson, and J. H. Kim, "Core: A real-time network emulator," in *Military Communications Conference, 2008. MILCOM 2008. IEEE*. IEEE, 2008, pp. 1–7.

[22] A. Botta, A. Dainotti, and A. Pescapè, "A tool for the generation of realistic network workload for emerging networking scenarios," *Computer Networks*, vol. 56, no. 15, pp. 3531–3547, 2012.

[23] J. Crowcroft and P. Oechslin, "Differentiated end-to-end internet services using a weighted proportional fair sharing TCP," *SIGCOMM Comput. Commun. Rev.*, vol. 28, no. 3, pp. 53–69, Jul. 1998. [Online]. Available: http://doi.acm.org/10.1145/293927.293930

[24] D. Damjanovic and M. Welzl, "MulTFRC: providing weighted fairness for multimediaapplications (and others too!)," *SIGCOMM Comput. Commun. Rev.*, vol. 39, no. 3, pp. 5–12, 2009.