# Rethinking the Development and Test of TCP

Paper # 44, 6 pages

## ABSTRACT

In this paper, we first reproduce a previous solution of building low-latency software rate limiters. We find a bug in DCTCP implementation, which explains the difference of the behaviors of the two approaches in the previous design. We fix the bug and get a rate limiter with the similar good performance. We analyze the root cause of the bug, and get two lessons. First, in TCP congestion control (CC) development, the decision-making whether a packet should be sent should be decoupled with the data plane execution of sending packets. Second, we propose to design a TCP testing framework which can make (more) complete testing for (new and existing) TCP variants. Our future work is to implement and apply this testing framework.

## 1 INTRODUCTION

Transmission Control Protocol (TCP) has evolved into many variants [6, 10, 13, 14, 18, 20] ever since it was invented, which adapts to various network infrastructures (e.g., the Internet or data centers) and application requirements (e.g., throughput or latency). While these TCP variants are assumed to be fully tested before being released to production, bug reports or failure news about TCP still occasionally appear, which usually affects TCP-based applications at a large scale. For example, Google researchers fixed a nearly decade old Linux kernel TCP bug in 2015 [1].

In this paper, we first reproduce an existing DCTCP-based work that improves software rate limiters [15]. The original Linux HTB would introduce queuing delay to traversing traffic, and the authors propose to apply ECN marking at the queue to reduce queue length. However, the authors observe that *marking data packets would cause TCP throughput to oscillate, in the contrast marking ACK packets can eliminate such oscillation*. The authors explained that marking data packets takes one RTT to feedback the congestion signal to the sender, during which time packets are already dropped; while marking ACK packets takes sub-RTT time, which avoids the problem. And the giant data frame size (caused TSO/GRO) would exacerbate the packet drop.

We make detailed experiments, instrumentation, and analysis, and argue that the performance difference is not caused by the one-RTT v.s. sub-RTT congestion signal feedback time but a bug at the receiver side. DCTCP has a design of delayed ACK and state machine to reduce the number of ACKs. When the state machine's state changes, DCTCP should "send immediate ACK"[6]; however, the "send immediate ACK" is mistakenly implemented which not only does not ACK the very last packet but also clears the delayed ACK timer. As a result, the sender side gets an ACK to reduce window and thus does not send data packets, and the receiver side does not send delayed ACK and waits for new data packets. The dead lock causes timeout at the sender side. And thus, the TCP throughput oscillates. When TSO/GRO are enable, the bug is easier to be triggered. We fix this bug, and show that the rate limiter's oscillation problem would disappear even with the design of data packet ECN marking.

We summarize two lessons. One lesson is about TCP congestion control development. TCP congestion control implementation should be decoupled from data path, which complies with the recent CCP work [17]. Furthermore, the abstraction interfaces between congestion control and data path should be well defined. In this way, congestion control developers can focus on congestion control algorithm essentials without dealing with low-level data path functions, which they may not be familiar with.

Another lesson is that a TCP testing framework should be built to make a complete test before a TCP is released. We analyze the parameters in a TCP testing, and design a testing framework. The framework consists of data plane agents to configure parameters on data plane entities and a control plane to generate testing plans. The framework can automate the TCP testing, which would be useful for TCP development and testing.

In this paper, we make the following contributions.

- We reproduce and refine an existing DCTCP-based low-latency software rate limiter.
- We fix a bug in the Linux DCTCP implementation.
- We summarize two lessons in TCP development and test.
  - In TCP CC development, decision-making and execution should be decoupled.
  - A TCP testing framework is needed.
- We propose the architecture of the TCP testing framework as the future work.

## 2 REPRODUCE A RATE LIMITER

We first elaborate the motivation, design, and evaluation of an existing software rate limiter. In [15], the authors observe that once a server is configured with hierarchical token bucket (HTB)[2], a TCP flow's RTT would increase significantly. The essential reason is that HTB would maintain a queue to buffer packets and dequeue packets according to the rate limiting; and the queue would introduce queuing delay for the traversing traffic. Thus, the authors would like to build a *low-latency rate limiter* (Figure 1).

The authors propose to apply TCP ECN marking to actively control the queue length so as to reduce the queuing delay. The design is to add a queue length threshold for the HTB queue — for each incoming packet, once the current queue length exceeds the threshold, the packet's ECN bit is marked.

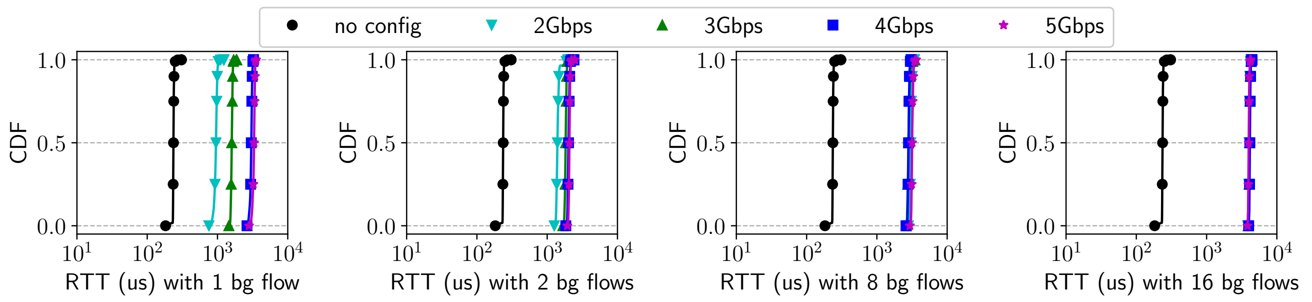

**Figure 1: Applying HTB in Linux would cause RTT to increase by 10X. This Figure is cited from [15].**

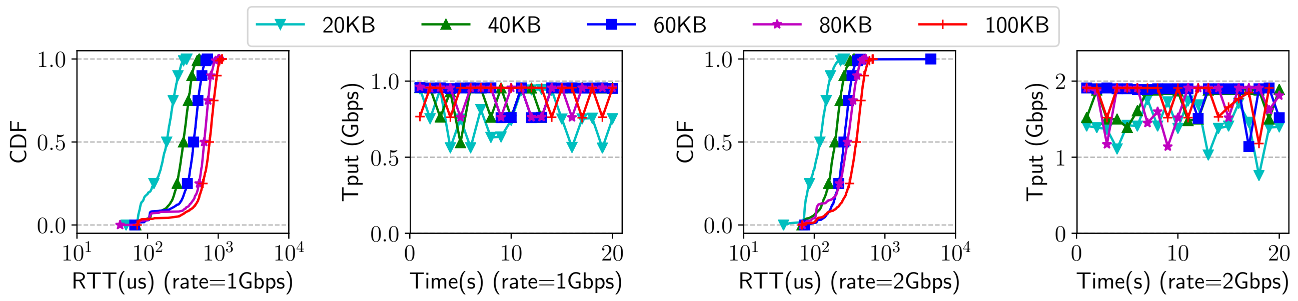

**Figure 2: Applying data packet ECN marking would cause throughput oscillation. This Figure is cited from [15].**

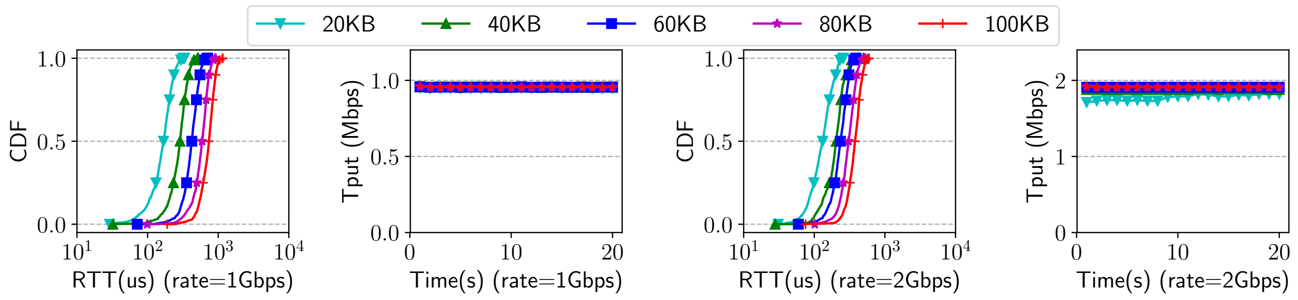

**Figure 3: Applying ACK packet ECE marking could solve the oscillation problem.**

On the end host, the sender side would react to the ECN marks and adjust the sending window; as a result, the queue length can be controlled.

There are two further approaches to implement such "ECN marking when the threshold is exceeded". The ECN marking can be performed on the **data packets**[1] or the **ACK packets**[2]. However, experiments show that ACK packet marking outperforms the data packet marking. Both approaches achieve the goal to reduce latency, but the data packet marking shows obvious *oscillation* while ACK packet marking shows no. We reproduce and compare these results in Figure 2 and 3.[3]

In [15], the authors argue that the reason of such oscillation is caused by the one-RTT congestion signal notification. In data packet marking, the congestion signal takes one round trip to arrive to the sender, during which many packets are already sent out and dropped; while in ACK packet marking, it takes sub-RTT time for an ACK to arrive at the sender, where the congestion signal feedback is more timely.

## 3 ROOT CAUSE

We analyze the difference of the performance of the two approaches above, and find and fix a bug in DCTCP implementation that causes the difference.

[1]Marking the ECN bit

[2]Marking the ECN Echo bit, i.e., ECE.

[3]The experiment settings is that there are one sender server and one receiver server, TSO/GRO is enabled on both sides, MTU is 1500 bytes, DCTCP is applied, and the Linux version is 4.6.0.

## 3.1 Analysis of the Previous Analysis

We argue that while ACK packet marking does solve the oscillation problem, the root cause is not the sub-RTT delay to deliver congestion signal. Considering the traditional on-switch solution, where ECN marking is applied to data packets, the congestion signal still takes one round trip to arrive at the sender, but there is no clue of similar TCP oscillation problem.

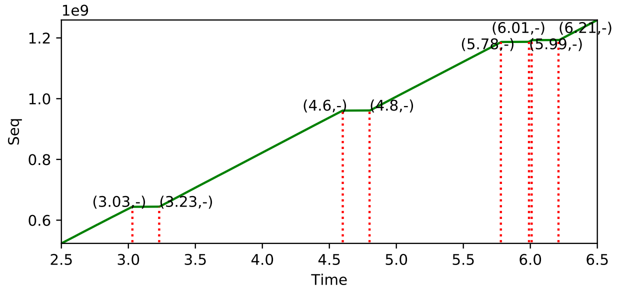

**Figure 4: Sequence number on sender side, 2Gbps link capacity, 30KB threshold**

We further verify the hypothesis that "one-RTT delay for congestion signal would cause more packet drops in this duration". We draw the sequence number of all traversing packets in Figure 4 (one of many experiments). Surprisingly, we do not observe sequence number drop (i.e., no retransmission), and there is no packet drop at all. But we observe the sender side frequently experiences time out (i.e., the line stalls without constantly increasing) which is 200ms in Linux.

## 3.2 Finding the Bug

With the observation that 1) there is not packet loss, and 2) time out happens; we suspect that the receiver side does not send ACK timely. We look into the code and finally confirm this assumption. It is essentially a complicated bug that happens when delayed ACK, DCTCP ACK state machine, TSO/GRO and ECN marking are configured together.

**The bug in DCTCP implementation.** On the DCTCP receiver side, ACK packets are triggered in two ways: 1) delayed ACK ("one cumulative ACK for every *m* consecutively received packets"[6] and a delayed ACK timer to avoid infinite waiting), and 2) ECN state machine's state change ("the states (in state machine) correspond to whether the last received packet was marked"[6]). Figure 5 shows how the state change in the state machine triggers an ACK packet.

However, the description "send immediate ACK with ECN=0/1" actually is incomplete in semantics, and it does not mention whether the packet that triggers this state change should be acknowledged. Suppose there is a delayed ACK *A* with $ECE = 1$ is pending on the receiver side at present and the next packet's ACK *B* with $ECE = 0$ arrives, triggering a state change, then "send immediate ACK with ECN=1" would be implemented in two ways: (1) send ACK *A* immediately but keep ACK *B*,

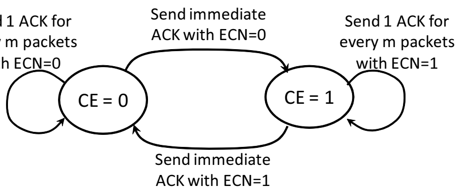
Send 1 ACK for every m packets with ECN=0
Send immediate ACK with ECN=0
Send 1 ACK for every m packets with ECN=1
CE = 0
CE = 1
Send immediate ACK with ECN=1

**Figure 5: Two state ACK generation state machine, cited from [6]**

then delayed ACK timer should be kept for *B*; (2) send ACK *A* and *B* immediately, then the delayed ACK timer can be cleared as there is no pending ACK delayed ACK left[4].

However, the Linux implementation makes a buggy mix of the two ways, it sends ACK *A* immediately, keeps ACK *B*, but clears the delayed ACK timer. Thus, the receiver side would never proceed to send ACK *B* out.

On the sender side, ACK *A* with $ECE = 1$ returns, causing the sending windows size to decrease (e.g., by a half[12] or proportionally to the number of ECEed packets). Although the ACK could move the window forward, the window's front side may not move forward due to the window size decreasing. And thus, the sender side is stuck at the states that no more data packets sending out and no more ACK packets arriving.

There is a *dead lock* where both sides are waiting for each other, and finally the sender side's RTO timer is triggered, and unacknowledged data packets are sent again; so the transmission restarts after 200ms.

**The risk to trigger the bug frequently.** In experiment, we also find that this bug is more frequent once TSO/GRO is enabled on both sides. In this case, data frames are large at both sides. At the sender side, once a giant data segment (64KB) is marked, it would cause a consecutive sequence of packets marked, leading to less number of state changes at the receiver side (and thus less ACKs); at the receiver side, multiple packets are merged as one segment, causing less number of data frames: when the state change is triggered, the triggering frame is more likely to have fewer successors to trigger a delayed ACK.

**A whole illustrative example.** We illustrate the bug using Figure6, which assumes ECN, TSO/GRO, and DCTCP are enable.

(1) In Figure 6, packets 100, 200, 300, 400, and 500 (identified by the sequence number) arrive at the receiver with ECN marks (CE=1)[5].

(2) The receiver GRO merges 100-400 but no 500, then the receiver acknowledge 100-400 (ACK=500) and start a delayed ACK timer for packet 500.

(3) The sender gets ACK 500 and moves window forward by 4 packets, and then another 4 packets 600, 700, 800, and 900 are sent.

[4]Clearing this timer is not mandatory, but could save resources.

[5]ECN is explicit congestion notification; CE is the "congestion encountered" bit in data packet header; ECE is the ECN-echo bit in ACK packet header; CWR is a "congestion window reduced" bit in ACK packet header.

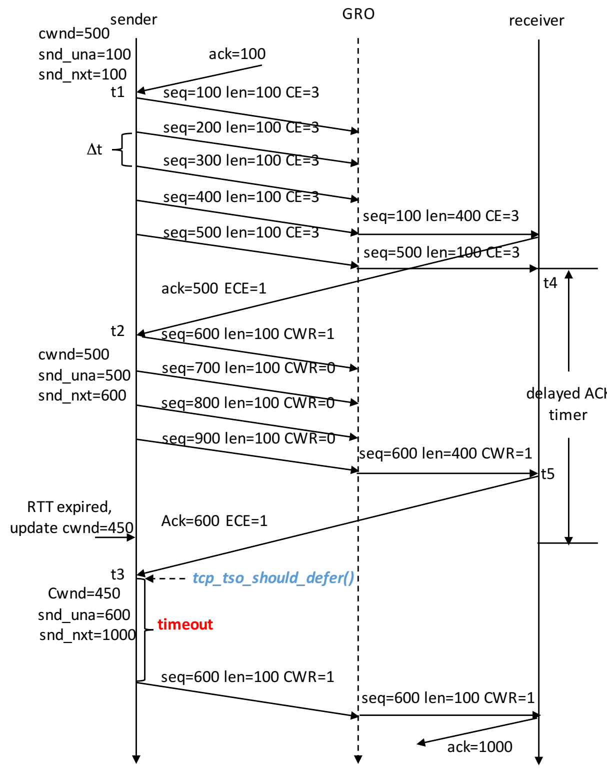

**Figure 6: A traffic pattern leading to timeout**

(4) Assume this time the packets 600-900 have no ECN marks (CE=0), the receiver GRO merges them and sends them as one packet; but a "state change" happens at the receiver this time.

(5) The state change would trigger an ACK for the packets *before* the change (i.e., packet 500), but hold the packet that triggers the change (merged packet 600-900).

(6) The newly triggered ACK (ACK=600) inherits ECE marking with ECE set to 1, and arrives at the sender. The sender sees two consecutive ACKs with ECE marked, reduces congestion window size, and does not send new packets.

(7) At the receiver side, the ACK of packet 500 is sent, and the ACK of merged packet 600-900 is pending. However, by calling a Linux kernel function `tcp_snd_ack()` to send packet 500's ACK, *all delayed ACK timers are cleared pointlessly*, including that of the merged packet 600-900.

(8) At this point, the sender does not send new packets, and the receiver has a delayed data packet (600-900) whose delayed ACK timer is cleared, thus, the sender would not get new ACKs to move congestion window forward. It would only wait for an RTO (200ms in Linux) and retransmit the unacknowledged packets in the congestion windows (packet 600).

## 3.3 Fixing the Bug

The essential reason of the bug is that Linux kernel function `tcp_snd_ack()` assumes "sending ACK" would drain all pending packets and thus "clear the pending delayed ACK timer"; but DCTCP would possibly "sending ACK of a few packets", and remaining packets' timer should be kept. We fix the bug in step (7), when `tcp_snd_ack()` is called and the pending packet's delayed ACK timer is cleared, we call `inet_csk_schedule_ack()` to reactivate its timer. The bug is fixed and we repeat the DCTCP experiment in Figure 2 and show the result in Figure 7. We observe high bandwidth saturation, low latency, and stable throughput in the same experiment setting. [6] Thus, both approaches of the low-latency software rate limiter design perform well.

## 4 A LESSON ON CC DEVELOPMENT

The root cause of the above bug is that congestion control developers use wrong *low-level* data path functions to control the transmissions of ACK packets. However, in essence, the congestion control algorithm mainly determines how *the sending rate or window* varies according to different congestion events, e.g., new ACK, ECN echo, and timeout. The *transmission* of each individual packet should not be the focus of congestion control algorithms but be that of the data path functions.

To improve the congestion control development, we argue that the implementation of congestion control algorithms should be decoupled from data path, which compiles with the recent perspective of CCP[17]. Following such principle, congestion control developers can focus on algorithm essentials and do not deal with low-level data path functions, which they may not be familiar with.

Furthermore, the abstraction interfaces between congestion control modules and the underlying data path should be well defined. Through these interfaces, congestion control modules can notify data path with the latest sending rate and window information while data path can keep delivering congestion events to congestion control modules.

## 5 A LESSON ON TCP TEST

Another lesson is drew that TCP is hard to be tested completely before releasing. And we propose to build a TCP testing framework.

### 5.1 Call for (More) Complete TCP Test

*Few TCP variants are completely tested.* Almost every TCP variant is evaluated after design and development, but their evaluations are usually a few "data points" in the complete picture of all possible TCP runtime environments. Moreover, few of their evaluations mentioned how to systematically explore all possibilities of the runtime environments.

*There are many parameters in TCP testing.* In the runtime, TCP interacts with the application atop it, the network stack below it, and the network between end hosts, which together comprise the TCP runtime environment (Figure 8).

[6] By the time when this paper is written, Google release another patch to fix this bug. Their patch does not reactivate the timer as we are, but immediately sends the delayed ACK of data packet that triggers the state change.[5]

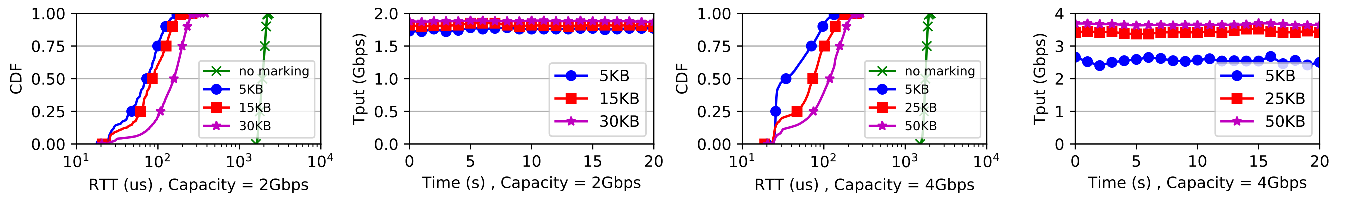

**Figure 7: After fixing the bug, data packet ECN marking achieves a throughput-stable rate limiter.**

However, few of the existing TCP variants are tested in all possible environments.

**Parameters of Workload Traffic.** Application workloads in TCP tests are usually generated by simple tools such as iperf, sockperf, or customized TCP client/server applications. These application layer workloads usually specify flow-level information such as flow size, arrival time, and/or duration, and flows are sent by the best effort. We propose the following flow characteristics can also be added in testing: sender/receiver buffer size, sending rate, burstiness, TCP socket options (e.g., blocking/nonblocking, keepalive, timeout, etc.).

**Parameters of Network Stack.** In the software part of the network stack, we propose the parameter of MTU. But for the hardware part (NIC), it is almost impossible to deploy all commodity NICs for testing; we suggest to use NICs with rich features like TSO/LRO (segmentation offload) as the parameter.

**Parameters of In-Network Devices.** For the network (devices) between end hosts, they interact with TCP by scheduling/forwarding packets, whose timing, delay, and drop of packets would affect TCP's control logic (e.g., congestion control). And similarly, a single testbed for TCP developers is almost not possible to have all kinds of devices and cabling. We abstract the testing parameters as ECN Marking, MTU, switch buffer, link capacity, loss rate, and background traffic.

*A TCP testing framework is needed.* Regarding to the bug we found which happens in the complicated environment (i.e., DCTCP, TSO/GRO, ECN, and HTB), we call for a plan to make more complete testing plans for TCP variants. And a testing framework is needed to automate this process.

## 5.2 A TCP Testing Framework

*TCP Performance Metrics.* We consider the following performance metrics to evaluate TCP.

**Throughput/bandwidth saturation.** Throughput of a TCP flow should be measured, and the measurement can be in different granularity, e.g., second, millisecond, microsecond. Bandwidth saturation is the ratio of achieved throughput over the link/path capacity, indicating the TCP's maximum capability to transmit data.

**Stability.** Throughput should not be only considered with its average value, but also its oscillation, i.e., its stability. The standard deviation of per-second / millisecond / microsecond throughput can be computed to represent a flow's stability.

**Latency.** Each packet's latency in delivery from the sender to the receiver should be measure. The average latency and the tail latency (e.g., 99.9 percentile) should be measured and evaluated.

**TCP internal variables (optional).** TCP has a lot of internal variables, e.g., timeout number, timeout length, retransmission times, and packet loss rate. Different from the three metrics above, they are not external properties shown to (and measured from) applications directly, but they can reflect whether the TCP is functioning correctly. We regard these internal variables as optional metrics to measure, because (1) they need instrument TCP to fetch[21], and (2) they are more useful in diagnosing than profiling application's performance.

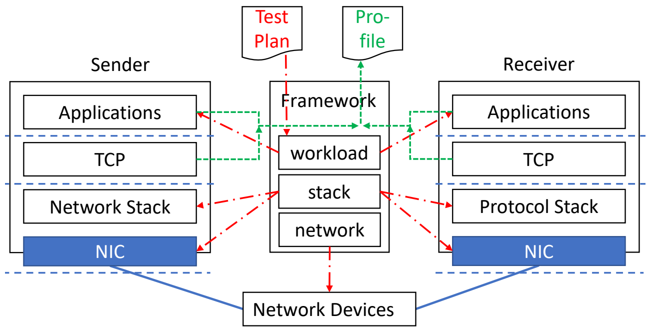

**Figure 8: The architecture of the testing framework.**

*Architecture.* (Figure 8) **Data Plane.** The entities that comprise a TCP testing environment are the sender/receiver application, the network stack, and the network devices. In each of them, there should be an agent to control the adherent parameters. And at the two ends there are traffic generators and TCP metric collectors.

**Control Plane.** The control plane should generate testing plans. A testing plan is a list of three tuple

`<entity, parameter, value>`,

which describes how an `entity` in the data plane is configured (`parameter` to be the `value`).

*Workflow.* The controller distributes a testing plan to all agents, traffic generators, and metric collectors, automates the testing, and collects the results. Special attention should be paid to distributed parameters such as MTU — they should be consistent across all entities.

## 6 RELATED WORK

**TCP CC Development.** TCP congestion control algorithms keep emerging [6, 10, 11, 14, 19]. To reduce the implementation bar of new congestion control algorithms, Linux has supported pluggable congestion avoidance modules [4], which allows users to implement customized congestion control algorithms in separate modules and plug these modules to enable them. Recently, CCP [17] explicitly proposes to move congestion control out of the data path.

**TCP Testing.** There exist a set of work about network workload measurement. Some of them describe distributions (temporal, spatial) of flow [6, 9]; when they are applied in TCP tests, intra-flow information (e.g., ECN marking, MTU) is missing. Another set of network measurement work is in packet-level[3, 22], which keeps packets timing information; this kind of traces, however, is hard to replay in TCP, because replaying them needs to coordinate packet timing in the application layer and ECN marking in the network layer, which is complex. In the evaluation of most recent transport protocols[6–8] the testing workloads are usually flows following a certain (measured) distribution, and few of them completely consider all the factors we list in this paper. DETER [16] can be used to test TCP in one specific environment repeatedly, but our proposal is to iterate all environments.

## 7 CONCLUSION AND FUTURE WORK

In this paper, we first reproduce an existing DCTCP-based solution to build a low-latency software rate limiter[15]. While we agree with the solution to solve the queuing delay in the previous work, but we disagree with the root cause of the oscillation problem in the previous design. The root cause is a bug in DCTCP implementation, where DCTCP's design of combining delayed ACK and state machine is mistakenly implemented using a Linux kernel function. The bug causes a dead lock of the sender and the receiver side which consequently causes timeout and throughput oscillation. We fix the bug, and our experiment shows that the bug fixing fill in the performance gap between the two approaches of the original design.

We learned two lessons from this experience. First, congestion control implementation should be decoupled from data path and the abstraction interfaces between them should be well defined. Second, with many parameters in the data plane, it is difficult to test TCP completely. We propose to build a TCP testing framework which consists of parameter tuning, traffic generation, and metric collection in the data plane and testing plan generation in the control plane. In the future, we expect to build the testing framework and make a complete test on existing and newly-built TCP variants.

## REFERENCES

[1] [n. d.]. http://bitsup.blogspot.com/2015/09/thanks-google-tcp-team-for-open-source.html. ([n. d.]).

[2] [n. d.]. http://luxik.cdi.cz/ devik/qos/htb/manual/userg.htm. ([n. d.]).

[3] [n. d.]. https://crawdad.org/. ([n. d.]).

[4] [n. d.]. https://lwn.net/Articles/128681/. ([n. d.]).

[5] [n. d.]. https://github.com/torvalds/linux/blob/master/net/ipv4/tcp_dctcp.c. ([n. d.]).

[6] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. 2010. Data center TCP (DCTCP). In *ACM SIGCOMM 2010 Conference*. 63–74.

[7] Mohammad Alizadeh, Abdul Kabbani, Tom Edsall, Balaji Prabhakar, Amin Vahdat, and Masato Yasuda. [n. d.]. Less is more: trading a little bandwidth for ultra-low latency in the data center. In *NSDI 2012*.

[8] Wei Bai, Kai Chen, Hao Wang, Li Chen, Dongsu Han, and Chen Tian. 2015. Information-Agnostic Flow Scheduling for Commodity Data Centers.. In *NSDI*. 455–468.

[9] Theophilus Benson, Aditya Akella, and David A Maltz. 2010. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*. ACM, 267–280.

[10] Lawrence S. Brakmo, Sean W. O'Malley, and Larry L. Peterson. [n. d.]. TCP Vegas: New Techniques for Congestion Detection and Avoidance. In *SIGCOMM 1994*.

[11] Neal Cardwell, Yuchung Cheng, C Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. 2016. BBR: Congestion-based congestion control. (2016).

[12] Kevin Fall and Sally Floyd. 1996. Simulation-based comparisons of Tahoe, Reno and SACK TCP. *ACM SIGCOMM Computer Communication Review* 26, 3 (1996), 5–21.

[13] Mario Gerla, Medy Y Sanadidi, Ren Wang, Andrea Zanella, Claudio Casetti, and Saverio Mascolo. [n. d.]. TCP Westwood: Congestion window control using bandwidth estimation. In *GLOBECOM 2001*.

[14] Sangtae Ha, Injong Rhee, and Lisong Xu. [n. d.]. CUBIC: A New TCP-friendly High-speed TCP Variant. *SIGOPS Oper. Syst. Rev.* ([n. d.]).

[15] Keqiang He, Weite Qin, Qiwei Zhang, Wenfei Wu, Junjie Yang, Tian Pan, Chengchen Hu, Jiao Zhang, Brent Stephens, Aditya Akella, et al. 2017. Low latency software rate limiters for cloud networks. In *Proceedings of the First Asia-Pacific Workshop on Networking*. ACM, 78–84.

[16] Yuliang Li, Rui Miao, Mohammad Alizadeh, and Minlan Yu. 2019. {DETER}: Deterministic {TCP} Replay for Performance Diagnosis. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*. 437–452.

[17] Akshay Narayan, Frank Cangialosi, Deepti Raghavan, Prateesh Goyal, Srinivas Narayana, Radhika Mittal, Mohammad Alizadeh, and Hari Balakrishnan. [n. d.]. Restructuring Endpoint Congestion Control. In *SIGCOMM 2018*.

[18] Kun Tan, Jingmin Song, Qian Zhang, and Murad Sridharan. [n. d.]. A compound TCP approach for high-speed and long distance networks. In *INFOCOM 2006*.

[19] Kun Tan, Jingmin Song, Qian Zhang, and Murari Sridharan. [n. d.]. A compound TCP approach for high-speed and long distance networks. In *IEEE INFOCOM 2006*.

[20] David X Wei, Cheng Jin, Steven H Low, and Sanjay Hegde. 2006. FAST TCP: motivation, architecture, algorithms, performance. *IEEE/ACM transactions on Networking* 14, 6 (2006), 1246–1259.

[21] Minlan Yu, Albert G Greenberg, David A Maltz, Jennifer Rexford, Lihua Yuan, Srikanth Kandula, and Changhoon Kim. 2011. Profiling Network Performance for Multi-tier Data Center Applications.. In *NSDI*, Vol. 11. 5–5.

[22] Qiao Zhang, Vincent Liu, Hongyi Zeng, and Arvind Krishnamurthy. [n. d.]. High-resolution measurement of data center microbursts. In *IMC 2017*.