

STMS: Improving MPTCP Throughput Under Heterogeneous Networks

Hang Shi and Yong Cui, ***Tsinghua University;*** **Xin Wang,** ***Stony Brook University;***
Yuming Hu and Minglong Dai, ***Tsinghua University;***
Fanzhao Wang and Kai Zheng, ***Huawei Technologies***

https://www.usenix.org/conference/atc18/presentation/shi

This paper is included in the Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC '18).

July 11–13, 2018 • Boston, MA, USA

ISBN 978-1-939133-02-1

Open access to the Proceedings of the 2018 USENIX Annual Technical Conference is sponsored by USENIX.

STMS: Improving MPTCP Throughput Under Heterogeneous Networks

Hang Shi
Tsinghua University

Yong Cui *
Tsinghua University

Xin Wang
Stony Brook University

Yuming Hu
Tsinghua University

Minglong Dai
Tsinghua University

Fanzhao Wang
Huawei Technologies

Kai Zheng
Huawei Technologies

Abstract

Using multiple interfaces on mobile devices to get high throughput is promising to improve the user experience. However, Multipath TCP (MPTCP), the de-facto standardized solution, suffers when different paths have heterogeneous quality. This problem is especially severe when the difference is the path latency. Our experimental results show that it causes the burst sending of packets from the fast path, which requires the in-network buffer to be big to achieve the full benefit of the bandwidth aggregation. In addition, it also requires bigger host buffer to fully utilize the fast path. To solve these problems, we propose and implement a new scheduler, which pre-allocates packets to send over the fast path for in-order arrival. Instead of relying on the estimation of network path condition, our scheduler dynamically adapts the MPTCP-level send window based on the packets acknowledged. Our evaluation shows that our scheduler can improve the throughput by 30% when the in-network buffer is limited, 15% when the host buffer is limited.

1 Introduction

There is a huge demand on network bandwidth with the rapid growth of network users and applications, as well as the emergence of bandwidth hungry applications such as multimedia streaming, cloud computing and virtual reality. To obtain high network throughput, a lot of recent interests have been drawn to exploit multi-path transmissions and aggregate the bandwidth through Multi-path TCP (MPTCP) [12]. As an example application, many wireless devices have two network interfaces, one to the local-area WiFi network and another to the wide-area cellular network. As wireless bandwidth is limited, a data stream can go through both networks to increase the transmission rate.

*Corresponding author

MPTCP is expected to be backward-compatible with conventional TCP and work with existing network components such as middle-boxes [26]. For the practical deployment of MPTCP, it is designed to be transparent to both applications and middle-boxes. From the perspective of applications, a single standard TCP is seen, whereas lower in the stack, MPTCP splits the data over multiple sub-flows. From the perspective of middle-boxes, each sub-flow is a normal TCP connection. MPTCP has already been implemented in Linux kernel [7] and used in iOS [4] and Giga Path in Korean Telecom [27]. Agache *et al.* [2] deploy MPTCP in the datacenter network to obtain better network utilization. Han *et al.* [15] apply MPTCP to improve the user experience on video streaming.

The core element of MPTCP design is the scheduler [28], which determines when and how to distribute packets to each sub-flow. MPTCP's default scheduler (minRTT) [26] sends packets through the available path with the smallest estimated Round-Trip Time (RTT). However, this scheduler does not take into account the path heterogeneity while there often exist different types and quality of paths in practical use and this is especially the case for wireless applications [17, 29]. The measurement of Alexa top-500 U.S. websites from Nikravesh *et al.* [24] shows that the difference in RTT between WiFi and Cellular paths is common and big. When RTTs in separate paths differ, the default scheduler will cause an out-of-order arrival of packets at the receiver side. Thus the memory requirements of MPTCP are much higher than those of conventional TCP. To alleviate this problem, opportunistic retransmission and penalization mechanism is proposed [26] and improved [25] along with the progress of the default scheduler.

Despite these efforts, we find that the complete gain from bandwidth aggregation of MPTCP is still far from being achieved. In our experiments conducted in the controlled lab environment (§ 2), we observe that when the host buffer is limited, the aggregation throughput is far smaller than two single-path TCP combined under

the same network and buffer settings. Sometimes it can only reach 40% the throughput of two single-path TCP together, which is even worse than a single-path TCP running over the best network interface.

Apart from the host buffer problem, we find that it also requires a big in-network buffer on the fast path to reach the full-bandwidth allowed by multiple paths. The in-network buffer requirement for the fast path is increased by 4X when using both paths compared with those for single-path TCP. Examining the problem carefully, we find that the default scheduler breaks down the ACK clocking [19] on the fast path, which gives rise to the burst sending of packets during fast sub-flow's transmission. Thus more in-network buffers are necessitated to hold the burst packets, otherwise the packets that cannot be stored have to be dropped, resulting in the throughput degradation on the fast path. This problem is more serious in the wireless networks where WiFi path is usually the fast one and has less buffer than that of the Cellular path [20]. Therefore when competing with single-path TCP, MPTCP is more likely to experience loss and the throughput will suffer. To the best of our knowledge, we are the first to identify the burst transmission pattern on the fast-path of MPTCP.

In this work, we propose a new scheduler to reduce both host buffer and in-network buffer requirements in MPTCP, called STMS (**S**lide **T**ogether **M**ultipath **S**cheduler). To solve the above two problems fundamentally, we need to reduce the out-of-order arrival. Our scheduler pre-allocates packets for the fast path and sends packets with larger sequence number through slow path so that packets can arrive at the receiver in order. This task appears to be straightforward, but it faces several challenges. As a matter of fact, there exists a "visibility gap" between the sender and the receiver. Therefore, it is hard for the scheduler that runs at the sender to ensure that packets arrive in order at the receiver. A sender can choose to utilize the measurement of path condition to schedule the packets, but the network condition is fluctuated in nature. Consequently, it is hard to measure the delay and bandwidth accurately especially in wireless networks. Even under stable network conditions, we can only obtain RTT but not one-way delay (OWD) in a practical distributed network. To address these challenges, we design an adaptation scheme that exploits the intelligent transmission of feedback signal *Data ACK* existing in MPTCP to dynamically schedule packet transmissions.

We implemented STMS as a Linux kernel module (§ 4) and evaluated it extensively in both emulated and real Cellular/WiFi networks. The results show that, compared with state-of-the-art schedulers [23, 25], STMS achieves significantly higher performance under a wide range of buffer/network conditions. We highlight some of results as follows:

- Under stable network conditions, STMS can achieve higher throughput under any buffer conditions. When the host buffer is extremely limited, STMS can fall back to using the single-path TCP, while the default scheduler still uses slow path of MPTCP and its fast path suffers from significant throughput degradation. In this case, our scheduler can improve the throughput as much as 400%. When the host buffer is small, STMS can bring 15% improvement over the default scheduler. When in-network buffer is limited, STMS improves the throughput as much as 30% due to the reduction of the burst.

- Under varying network conditions, STMS also performs well, and brings 8% to 40% throughput improvement. Our adaptive scheduling scheme is reactive to network condition changes.

- In real world test, STMS can reduce the file downloading time by 20% even when the host buffer is big enough, proving that the limited in-network buffer does exist in real network and our scheme effectively alleviates the problem.

Overall, our results show that by strategically scheduling packet transmissions to reduce the out-of-order arrival, STMS can significantly improve the throughput of MPTCP under heterogeneous networks. The rest of the paper is organized as follows. Firstly, we analyze the reason that lead to the throughput degradation when using the default scheduler in § 2. Then in § 3, we present the design and analysis of STMS. Next we introduce our implementation of STMS in § 4. We further present our performance evaluations in a controlled-lab environment in § 5, and the results from the real-world test in § 6. Finally, related work is discussed in § 7 and we conclude the paper in § 8.

2 Background and Motivation

In this section, we first identify and analyze the problem associated with the in-network buffer. Then we demonstrate that host buffer problem still remains unsolved and discuss why the existing solution is still inadequate. We support our analyses with experiments conducted in the controlled lab environment.

2.1 Controlled experiment setup

In our experiment setup in fig. 2, we use two PCs running the version 0.92 kernel of MPTCP [7] as the client and server respectively. The client has two interfaces, and the server has one interface. Under this topology, MPTCP will establish two sub-flows. We use the decoupled TCP



(a) t = δ (b) t = RTT_f (c) t = RTT_s

Figure 1: Out-of-order arrival and burst transmissions on the fast path due to the use of default scheduler. Green (lighter) packets are sent/received through the fast path, while blue (darker) ones are sent/received through the slow path.

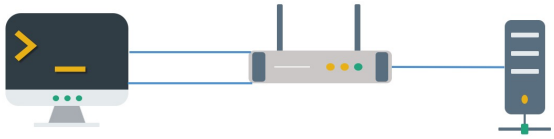


Figure 2: Topology setup

congestion control to obtain the best bandwidth aggregation effect [9]. Ethernet is used to simulate WiFi and LTE to avoid the interference of the wireless network. Client and server are connected through a router running OpenWrt. We use tc [30] in OpenWrt to regulate the bandwidth and latency of the two paths. The bandwidths of both paths are set to 30 Mbits and the loss rate is set to 0.01%. The in-network buffer is set to 50ms for Wifi path unless specified otherwise and 500ms for LTE path based on [20]. Both receiving and sending buffers are set to the Linux default size (6MB) unless specified otherwise. Only the RTT of slow path varies. In each network setup, we run iPerf 3 [18] 90s five times to measure the throughput.

2.2 Big in-network buffer requirement

Guido *et al.* [3] points out that it becomes more and more difficult to design the router with the buffer size equal to the bandwidth-delay product as the link speed increases. The router buffer is limited especially in the bottleneck of path. However, we find that the aggregated throughput is subject to the in-network buffer on the fast path as shown in fig. 3. For conventional TCP, 8ms network buffer (30KB) is enough to reach the full bandwidth. However, for MPTCP under different RTT paths, the in-network buffer requirement increases as much as 4X to fully unleash the power of the bandwidth aggregation. The bigger the difference of RTT between two paths, the bigger the in-network buffer of the fast path is needed.

When there is a space in the congestion window, the default MPTCP scheduler sends a set of packets with the smallest sequence numbers on the path with the smallest estimated RTT. This approach will produce the out-of-order arrivals at the receiver, as is elaborated in the following example (fig. 1). At the time $t = 0$, we assume the fast path is unavailable while the slow path has space,

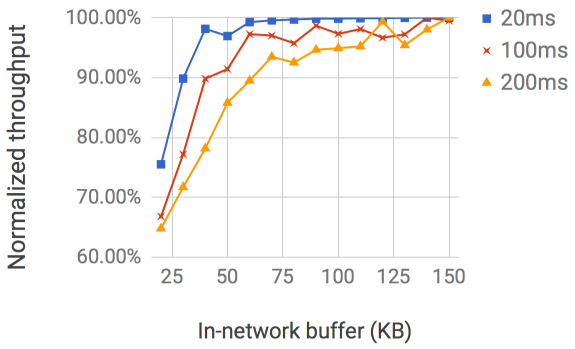
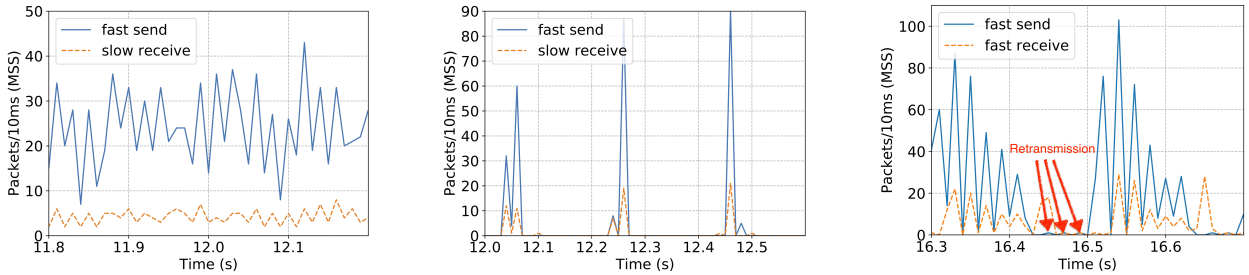


Figure 3: MPTCP throughput degradation when in-network buffer is limited

then packets 1-10 will be sent on the slow path. Later on at $t = \delta$, the fast path becomes available, packets 11-30 will be sent on the fast path. After the round-trip time of the fast path RTT_f, packets 11-30 arrive at the receiver but packets 1-10 do not, and the send/receive window is blocked as shown in fig. 1b.

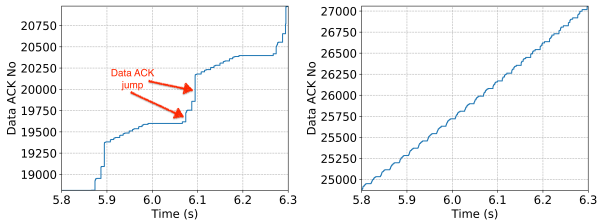
TCP has the delayed ACK mechanism [6] to avoid the overhead of sending ACK packets. It works as follows. Upon receiving a data packet, if it is in order, *i.e.,* the right edge of the receiving window advances, the receiver can choose to delay the sending of ACK hoping to piggy-back the ACK with other packets to send in the reverse direction. Nevertheless, RFC 1122 [6] suggests that each ACK acknowledge at most two packets regardless of whether the delayed ACK mechanism is used, *i.e.,* upon receiving two successive packets, an ACK must be sent. Thus, during the congestion avoidance phase, upon receiving one ACK, the sender's send window can have the space for at most two packets so that it can send at most two packets at a time. This is also known as ACK clocking.

In MPTCP, the semantics of the TCP send window is generalized. Instead of maintaining a separate window for each sub-flow, a single buffer pool is shared by all sub-flows at the MPTCP-level to avoid deadlock [26]. To remain compatible with TCP, MPTCP needs a separate ACK for MPTCP-level send window, called Data ACK. The delayed ACK mechanism of conventional TCP is adapted to Data ACK. When receiving data packets in-order on the MPTCP-level, Data ACK will still be



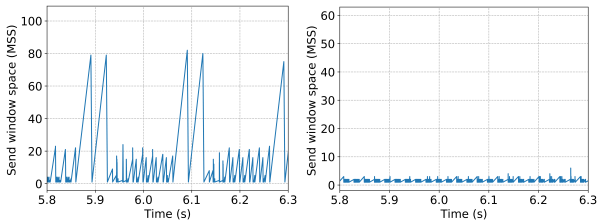
(a) $RTT_f = 20ms, RTT_f = 20ms$ w/o PR (b) $RTT_f = 20ms, RTT_s = 200ms$ w/o PR (c) $RTT_f = 20ms, RTT_s = 200ms$ w/ PR

Figure 4: Burst sending pattern of fast path



(a) $RTT_f = 20ms, RTT_s = 200ms$ (b) $RTT_f = 20ms, RTT_s = 20ms$

Figure 5: The Data ACK of fast path.



(a) $RTT_f = 20ms, RTT_s = 200ms$ (b) $RTT_f = 20ms, RTT_s = 20ms$

Figure 6: Send window free space jitter when RTT differs

generated every other packet. However, when receiving out-of-order packets on the MPTCP-level, it will not send a duplicate Data ACK immediately since out-of-order packets on the MPTCP-level is a normal behavior especially when paths are heterogeneous. The Data ACK won't be sent until the packets from the slow path reach the receiver, *i.e.,* the hole is filled. This Data ACK will acknowledge many packets sent from the fast path at the same time, thus fast path can send many packets at once, leading to the burst sending. In a nutshell, the out-of-order arrival of packets breaks down the ACK clocking effect of fast sub-flow, causing the burst sending behavior. This is shown in fig. 1c. At t = RTT_s, packets 1-10 arrive at the receiver and packets 1-30 are acknowledged to the sender. So both the send and receive window will progress with a large step, and packets 31-50 are sent from the fast path in a burst. Simply removing the delayed ACK mechanism can not solve the problem, since the MPTCP-level send window progress is still blocked by the late arriving packets from the slow path.

We conduct a simple experiment to demonstrate the burst transmission pattern of the fast sub-flow.

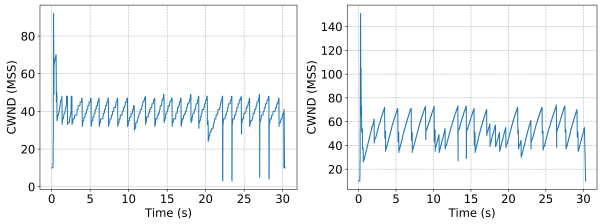
First we analyze the progress of Data ACK of the fast path from the trace file. The result is shown in fig. 5. Only when packets from the slow path arrive, the Data ACK can make a progress with a large step (fig. 5a). When RTT is the same, no sudden Data ACK progress happens. Then we check the free space in the send window of the fast path. We record the free space of send window when the sub-flow receive ACK (fig. 6). When RTT is the same, the ACK clocking is maintained like the conventional TCP (fig. 6b). However, when RTT over two paths significantly differs, the ACK clocking of the fast path is broken down. Consequently, MPTCP-level send window is stalled until the packets from the slow path arrive. As a result, the free space of the send window of the fast sub-flow will accumulate as we see in fig. 6a.

The sudden progress of MPTCP-level window and the big send window space of fast path will lead to the burst packet transmissions on the fast path as we see in fig. 4b. Compared to the sending behavior of the fast path when RTT is similar (fig. 4a), the fast sub-flow clearly demonstrates an on-off transmission pattern that leads to the burst of the fast sub-flow. When the network buffer is not big enough, the loss will happen and the Congestion window (CWND) is capped to a small value as we see in fig. 7a. When there is a difference between RTT on the two paths, the CWND of the fast path is capped around 40, compared to 60 when RTT values of two paths are the same. Note that there is usually a large space in fast path's CWND when RTT is different, so the area below the CWND line is actually bigger than the total throughput.

We also verify the loss rate under different in-network

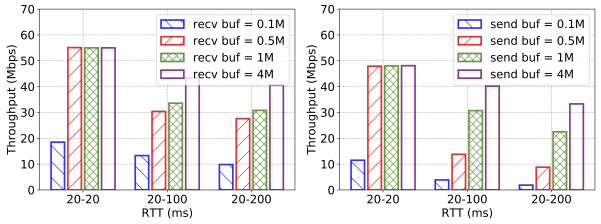
Table 1: Loss rate of fast path under different in-network buffer setting

in-network buffer/KB	observed loss rate	Fast path in MPTCP/Mbps	Single TCP/Mbps	Utilization
30	0.05%	12.1	28.4	42.76%
60	0.02%	20.8	28.4	73.50%
90	0.02%	25	28.4	88.34%
150	0.01%	26.5	28.4	93.64%



(a) $RTT_f = 20ms, RTT_s = 200ms$ (b) $RTT_f = 20ms, RTT_s = 20ms$

Figure 7: CWND of fast sub-flow is capped when in-network buffer is limited and RTTs are different.



(a) Receive buffer (b) Send buffer

Figure 8: MPTCP throughput degradation when host buffer is limited

buffer configurations (table 1). Note that this loss is more detrimental to the throughput than the random packet loss since each time CWND grows to certain value, the packet burst will exceed the in-network buffer size and the loss will happen. As a result, the throughput of the fast path is significantly limited. As shown in table 1, to reach the same throughput as that of single-path TCP, the buffer for two-path MPTCP has to be increased by 4X from 30KB to 150KB.

2.3 Big host buffer requirement

Another problem of MPTCP when RTTs are different is the big host buffer requirement. Let us denote the bandwidth of fast and slow sub-flow as B_f and B_s respectively. The one-way delay (OWD) values of both paths are denoted as OWD_f and OWD_s respectively. The RTTs of both paths are denoted as RTT_s and RTT_f. Raciu *et al.* [26] derives that the default scheduler buffer requirement is:

$$Buf(default) = (B_f + B_s) \times RTT_s \quad (1)$$

From eq. (1) we can see that the key to reducing the buffer requirement is to reduce the effective RTT of the connection *i.e.,* to acknowledge packet as soon as possible so that the buffer can get freed.

Raiciu *et al.* [26] proposed the penalize and opportunistic retransmission mechanism (PR) to deal with the host buffer problem. When it detects the left edge of the send window blocks the sending of packets, it will retransmit the packet from the fast path. To avoid constant retransmissions, it will penalize the slow sub-flow by halving the CWND of slow sub-flow. This approach does improve the throughput when the receiving buffer is limited because the blocked packets will get retransmitted and acknowledged through the fast path. Thus the effective RTT of MPTCP connection can be reduced to $OWD_s + OWD_f$ ideally. As shown in fig. 4c, the packets marked get retransmitted and the new Data ACK goes back through the fast path so many packets can be sent. Due to its opportunistic nature, the PR scheme can not always reduce the RTT to the optimal value. Moreover, the retransmission wastes the bandwidth. fig. 4c also shows that the retransmission does not change the fast path's burst transmission pattern, as the Data ACK coming back through the fast path will still acknowledge many packets. Hence the in-network buffer requirement issue remains unsolved which is also shown in the in-network buffer requirement measurement. Enabling retransmission doesn't improve the throughput at all.

Despite the PR approach, the throughput of MPTCP is still unsatisfactory as shown in fig. 8. When evaluating the receiving buffer, the sending buffer size is set to the default value of Linux which is 6 MB (Big enough under our network setting). As we can see in fig. 8a: the bigger the RTT difference of two paths, the more receiving buffer is needed to get full bandwidth aggregation effect of MPTCP. The same conclusion goes to the sending buffer as is shown in fig. 8b.

Actually, the burst sending pattern can be fixed by adding traditional pacing to the fast path. Linux has supported pacing in tc qdisc fq [13] since the version 2.4. However the pacing rate argument needs to be manually tuned according to the bandwidth of the network path. New congestion control algorithm Bottleneck Band-

width and Round-trip propagation time (BBR) [8] incorporates the pacing and can set the pacing rate equal to the measured bottleneck bandwidth automatically. However the congestion control of MPTCP needs to be fair with the conventional TCP. Many active research efforts [5, 21, 31] have been put into developing the coupled congestion control algorithms of MPTCP, but there is no coupled BBR congestion control for MPTCP yet. To put it another way, this approach is not congestion control agnostic. Besides, the pacing can not solve the big host buffer requirement issue. Pacing works only on sub-RTT level and thus the packet sent from the slow path will still arrive later than packets sent from the fast path. We verify that by adding pacing manually and it turns out that the throughput is not improved at all.

3 Design

The root cause of the throughput degradation is the stall and sudden progress of MPTCP-level send window. To solve both host buffer and in-network buffer problems, we need to restore the ACK clocking for MPTCP. To achieve this, packets need to arrive in order at the receiver.

In this section, we first present our algorithm design, then derive the size required from the host-buffer for no-blocked transmissions, and finally compare the transmission latency of different schemes.

3.1 STMS algorithm

We propose STMS to schedule the packets strategically so that they arrive in order. This solution is congestion control agnostic which allows for separate evolving of congestion control algorithm and scheduler algorithm.

Scheduler algorithm The core idea of our scheduler is to buffer packets for the fast sub-flow and assign packets with larger sequence numbers to the slow sub-flow so that they arrive in order. The running process of our algorithm is shown on fig. 9 and algorithm 1. The scheduler runs when at least one of paths is available to send packets. The fast sub-flow always sends the packets with the smallest set of sequence numbers in the buffer. As illustrated in fig. 9a, the slow sub-flow sends packets with bigger sequence numbers. Rather than taking packets whose numbers are right after those transmitted on the fast path, it leaves a sequence gap for the fast path to send the corresponding packets in the future. By the time the packets from the slow path arrive, all packets from the fast path which have smaller sequence numbers should have already arrived (fig. 9b). Since packets arrive in order, the normal ACK clocking is ensured, so there are no burst transmissions on the fast path and the send/receive window will not be blocked.

Algorithm 1 Slide Together Scheduler

```
1: procedure ST_SCHEDULE(unsentPackets)                ▷
   Scheduler runs when one of sub-flow is available
2:     if Fast sub-flow has space in send window then
3:         Fast sub−flow ←unsentPackets[0]
4:     elseSlow sub-flow has space in send window
5:         Slow sub−flow ←unsentPackets[Gap]
6:     end if
7: end procedure
```

The key parameter of the scheduler algorithm is *Gap*, which is the number of packets pre-allocated for fast path to send. The efficiency of the scheduler algorithm depends on the accuracy of the gap value. Any deviation from the true value will cause out-of-order arrival of packets.

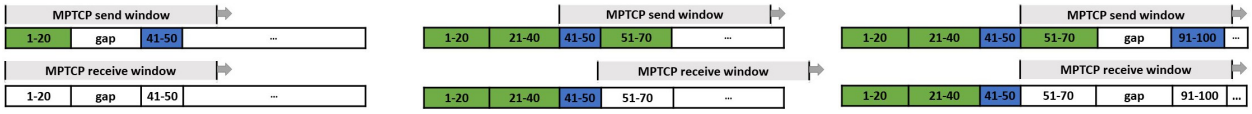
The naive way to get the gap value is to utilize the measurement of path conditions. If we can measure network conditions accurately, then we can derive the true gap value in the following way. It takes $OWD_f + \frac{Gap}{B_f}$ for all packets in the gap to arrive at the receiver through the fast path, where B_f is the bandwidth of the fast path. This should be equal to OWD_s so that the first packet from the slow path arrives at the same time as packets from the fast path. Then we have the true Gap value:

$$True_Gap = B_f \times (OWD_s - OWD_f) \tag{2}$$

However, the naive solution has two fundamental flaws. One is that the one-way delay of path can not be measured accurately. We can not assume the uplink delay and the downlink delay are the same on both paths. If we modify the protocol to carry the one-way delay information, it may cause other compatibility problem with middleboxes [16]. The other one is that the bandwidth of the path can not be measured accurately, especially when the in-network buffer is limited. Because of the burst sending pattern of fast sub-flow, it can never reach the actual available bandwidth. So we design the feedback-based gap adjustment scheme to adjust the value of gap more accurately and quickly.

Key insight: Every out-of-order packet in the receiver will generate duplicate Data ACK or burst Data ACK.

What STMS actually does is moving stalled packets from the receiver to the sender (*i.e.,* the packets inside gap). The out-of-order packets will be acknowledged at one time when packets from the slow path arrive to fill in the hole. The number of packets acknowledged by Data ACK reflects the degree of out-of-order arrival of packets. Accordingly we can use this as the feedback signal to adjust the gap value. Since Data ACK is presented in MPTCP, our scheme remains compatible with current MPTCP. In addition, this scheme does not require



(a) t = 0 (b) t = $2RTT_f$ (c) t = RTT_s

Figure 9: Demonstration of STMS, with green and blue for packets over fast and slow paths respectively. Let $RTT_s = 3RTT_f$ and assume the uplink delay and downlink delay are symmetric, then we have $gap = CWND_f$ according to eq. (2). Note that slow path always send packets with sequence numbers bigger than those of the fast path.

any modification at the receiver, which further eases the deployment process.

How to adjust The gap adjustment algorithm is shown in algorithm 2. Let $delta_gap$ and $adjust_interval$ be the gap adjustment step and interval. When the gap is smaller than the true gap value, the packets from the slow path arrive late and the send window of MPTCP-level will be stalled by packets sent from the slow path. Therefore the left edge of the send window is determined by unacknowledged packets from the slow path. Symmetrically, when the gap is bigger than the true gap value, the left edge of the send window will be determined by the packets sent from the fast path. Each time we receive a Data ACK, we first calculate how many packets this Data ACK acknowledged ($data_acked$). If the $data_acked$ is bigger than 2, we will adapt the gap value. We check the packet of the left edge of the send window. If the packet is the first one sent from the slow path, $delta_gap = data_acked$; otherwise, $delta_gap = -data_acked$. We use the Exponentially Weighted Moving Average (EWMA) of $delta_gap$ over $adjust_interval$ to adjust the gap value. The $adjust_interval$ is a tunable parameter, which determines how fast the gap adjustment can react to the network change. Setting it too small will cause the gap overshoot and oscillation since the previous gap adjustment has not taken into effect yet. However, setting it too big leads to the slow convergence time.

Algorithm 2 Gap Adjustment Algorithm

```
1:  procedure GAP_ADJUST(data_acked)        ▷ This function gets called when receiving Data ACK
2:      if data_acked > 2 then
3:          send_una ← left edge of MPTCP-level send window
4:          if send_una was sent from slow path then
5:              delta_gap = data_acked
6:          else send_una was sent from fast path
7:              delta_gap = −data_acked
8:          end if
9:      end if
10:     gap+ = EWMA(delta_gap, adjust_interval)
11: end procedure
```

3.2 Analysis of host buffer size requirement

At a first glance, buffering packets for the fast path may require a big buffer on the sender side. However, we can prove that when both paths are fully utilized, the send buffer requirement is actually less than that of the default scheduler without PR (eq. (1)).

When using STMS the send buffer consists of three parts:

1. sent but unacknowledged packets from the fast path: $B_f \times RTT_f$
2. sent but unacknowledged packets from the slow path: $B_s \times (OWD_s + OWD_f)$ (Data packet is sent through the slow path, but ACK returns from the fast path).
3. buffered packets for the fast path *i.e.,* $True_Gap$ (eq. (2))

By adding these three parts together, we get the buffer requirement of STMS:

$$Buf(STMS) = (B_f + B_s) \times (OWD_S + OWD_f) \quad (3)$$

This is smaller than $Buf(default)$ (eq. (1)). It also reveals that STMS reduces the effective RTT of the MPTCP connection to $OWD_s + OWD_f$, which is the smallest RTT when both paths are utilized. Thus our STMS can reduce the send buffer requirement.

If we take into consideration the opportunistic retransmission, then in the ideal case, upon receiving the late arrival packet from the slow path, the Data ACK of the retransmitted packet will go back through the fast path. Therefore the effective RTT of the MP connection is reduced to $OWD_s + OWD_f$, which is the same as STMS and the buffer requirement is also the same.

When the host buffer is between [$Buf(STMS)$, $Buf(default)$], both retransmission and STMS can improve the throughput. But STMS can always achieve the optimal throughput by ensuring RTT of MP connection to be the minimum.

If the host buffer is smaller than $Buf(STMS)$, neither STMS nor default scheduler can get the full bandwidth

aggregation. In this case, STMS will prefer to use the fast path. The slow path will be used only if it will not cause the blocking. The buffer requirement to take advantage of the use of the slow path is:

$$\begin{aligned} Buf(fallback) &= RTT_f \times B_f + Gap \\ &= B_f \times (OWD_s + OWD_f) \end{aligned} \quad (4)$$

When the host buffer is between [$Buf(fallback)$, $Buf(STMS)$], STMS will use the fast path first, as the slow path will not block the fast path. However, for the default scheduler, the slow path will get the frequent use, which would trigger the retransmission of packets. This will further lead to the goodput degradation and the big end-to-end latency.

What if the host buffer is even smaller than $Buf(fallback)$? Then STMS will fall back to the single TCP over the fast path. But the default scheduler will still send some packets from the slow path, which pushes the effective RTT of MPTCP connection to at least $OWD_s + OWD_f$. Thus the throughput will be even worse than the bandwidth B_f allowed by the fast path alone. Actually, in this case, falling back to the single-path TCP is the optimal choice.

So our scheduler can get the optimal throughput across all range of host buffer sizes.

3.3 Analysis of latency

It seems that STMS will cause the inflation of transmission latency because it holds packets in the gap longer than the default scheduler. However, it also reduces the time duration for the packets to be stalled in the receiver. Using both types of scheduler, the end-to-end latency of packets sent from the slow path is OWD_s. The latency of the fast path is $OWD_f + Delay(Stalled)$. For each packet sent from the fast path, it is either stalled at the receiver buffer or held at the send buffer. $Delay(Stalled)$ remains the same. Therefore STMS does not increase the average end-to-end latency of packets.

4 Implementation

We implement STMS in the Linux kernel based on MPTCP version 0.92 from [7]. The algorithm 1 is implemented as a scheduler module.

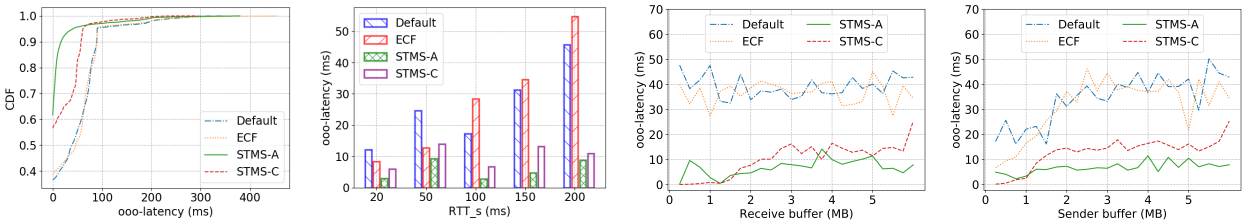
The MPTCP scheduler will run when two types of event happen. The first type of event happens when ACK returns from one of the sub-flows, which means there will be space in the sub-flow send window. The second type of event happens when application sends more data. The scheduler makes the decision every data segment. For each segment pushed in by the application, the scheduler will determine which sub-flow to send the packet. This framework of scheduler limits how we can implement our scheduler, since we can not access an arbitrary segment inside the send buffer. To remain compatible with this framework. We implement our scheduler as follows.

When the scheduler picks the next segment to send, we first check if the fast path is available, *i.e.,* there is space in the send window. If the space is available, then send the packet over the fast path; otherwise, we check if the slow path is available. If it is available, we find the packet according to the gap value, *i.e.,* jump across the gap packets to find the packets to send over the slow path. If the packet does not exist, that means either the packet is out of the right boundary of the send window or the application has not pushed enough data yet, in either of which case we skip this round of scheduling. Note that we need to mark the packets sent from the slow path so that we can skip the out-of-order packets when finding the next packet to send from the slow path. To avoid traversing the send buffer from the beginning each time, we save the pointer of the last send packet of the slow path as the beginning point for the next search.

We implement two variants of STMS: STMS-C ("Calculation") and STMS-A ("Adjustment"). They both preallocate packets for the fast sub-flow so that packets can arrive in-order at the receiver side (§ 3.1). They differ in how they obtain the gap value. Each time a packet is sent from the slow path, STMS-C extracts the bandwidth estimation of smoothed RTT from subflow TCP's algorithm and calculates the gap value (assume the uplink and downlink delay are symmetric). For STMS-A the Data ACK process function is modified to calculate $delta_gap$ according to algorithm 2.

5 Evaluation in a controlled lab environment

In this section, we test both STMS-C and STMS-A in a controlled lab setting, which allows us to evaluate the performance across a wide variety of network conditions. We compare our scheduler with the default scheduler with PR (denoted as Default thereafter) and ECF [23]. ECF uses the send buffer length to estimate the flow complete time(FCT) of using each path. If using slow path will cause inflation of FCT, it will wait for fast sub-flow. However, for elephant flow, the send buffer is full for most of the time. Only when flow is about to finish, the send buffer length can be small enough to wait for fast sub-flow. Besides, when calculating the FCT, ECF does not take into account the one way delay thus it is not able to achieve accurate in-order arrival. STMS schedules the packets out-of-orderly to achieve the in-order arrival regardless of the send buffer status so STMS can outperform ECF. For apples-to-apples comparison, we port the ECF



(a) OOO delay distribution (b) Varying slow path latency (c) Varying receive buffer (d) Varying send buffer

Figure 10: Out-of-order latency of different schedulers

code [10] to the same MPTCP version as our scheduler. The experiment setup is the same as § 2.1. We tune the $adjust_interval$ of STMS-A to $\frac{RTT_s+RTT_f}{2}$ according to the analysis in § 3.1. The parameter of ECF is chosen as the same as [23].

5.1 Microbenchmarks

We first focus on some micro-benchmarks to see whether our scheduler can accomplish the design goal.

5.1.1 Reducing the out-of-order arrival at the receiver side

We first investigate whether our scheduler can achieve the in-order arrival at the receiver side. We use the out-of-order (OOO) delay as the metric. The OOO delay of a packet is defined as the time difference between when a packet arrives at the receiver to when the packet can be submitted to the application (*i.e.,* all packets with the smaller data sequence numbers have already arrived).

fig. 10a shows the CDF of the OOO delay of each packet with different schedulers. STMS-C and STMS-A can both achieve smaller OOO delay than Default and ECF. The largest OOO delay is around 300ms. Default effectively pushes the OOO delay of most packets sent from fast path to OWD_s. ECF sends tail packets out-of-orderly so it can reduce OOO delay for those tail packets. However since we transmit many packets for a test, this delay reduction is negligible. STMS-A can effectively push the OOO delay close to zero.

We vary the latency of slow path and calculate average of OOO latency. The result of one experiment is shown in fig. 10b. When paths become more heterogeneous, both Default and ECF have larger OOO delay. However STMS-A can keep its average OOO delay at a very small value. When RTT of two paths are similar(20ms and 50ms), ECF, STMS-C and STMS-A get close performance.

We also test the OOO delay under different host buffer sizes. The result is shown in fig. 10. It demonstrates that both STMS-A and STMS-C can effectively reduce the OOO delay regardless of the host buffer size, and the gain is larger when the host buffer sizes are larger with more packets stalled at the receiver. For ECF, only when the send buffer is very small, it wait for fast sub-flow to reduce the OOO delay.

5.1.2 Reducing the burst on the fast path

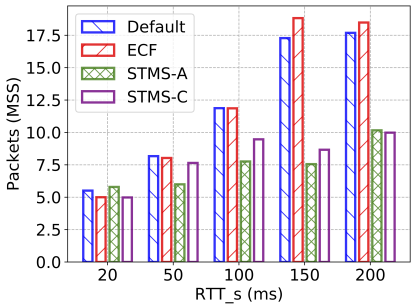


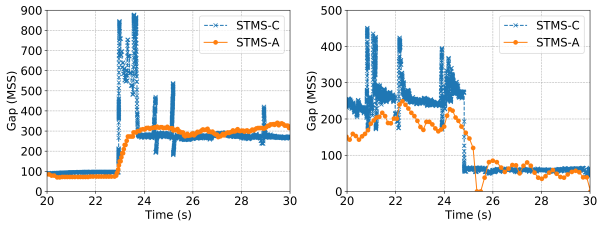
Figure 11: Burstness of all four schedulers under different path latencies

We now study whether our scheduler can reduce the burst of the fast path thus the in-network buffer requirement accordingly. We print the CWND free space of the fast sub-flow when it receives ACK. Since all schedulers try to fill this free space, the peak value of CWND free space reflects the burst of fast path. The average CWND free space throughout the running time is used as a metric of burstness of fast subfow. The result is shown in fig. 11. The trend is the same as fig. 10b. Our scheduler can reduce the burstness of the fast path and makes it close to that of the single-path TCP.

5.1.3 Gap adjustment is reactive to network change

To understand how STMS-A handles dynamic network conditions, we change the network conditions in the middle of MPTCP flow and record the gap value changes around the condition changing point. Recall that the true gap value is calculated using eq. (2). It is affected by the accuracy of the measurement of the fast path bandwidth and the one-way delay of both fast path and slow path. We choose to change the latency of the fast path and slow

path to demonstrate how our Gap adjustment algorithm reacts to the network change. In fig. 12a, the latency of the fast path changes from 20ms to 5ms. Therefore there will be many ACK packets and the fast path can send a lot of packets out which leads to the wrong estimation of the bandwidth. Thus, we see the peak of the gap calculated value. However, our gap adjustment algorithm can converge to the new value smoothly. In fig. 12b, the latency of the slow path changes from 200ms to 100ms. Again STMS-A converges to the new value smoothly and fast.



(a) Gap increase (b) Gap decrease

Figure 12: Gap adjustment is reactive to network change

5.2 Macrobenchmark: improving aggregated throughput

We then investigate how our scheduler improves the aggregated throughput under different buffer sizes setting.

Stable network condition We begin by investigating whether our scheduler improves the throughput when the network condition is stable. fig. 13 shows the result. When the in-network buffer is limited, our scheduler can improve the throughput by about 30% compared to Default and ECF. When the host buffer is extremely limited, our scheduler falls back to single path TCP and outperforms Default as analyzed in § 3.2. When the host buffer is big enough(>4MB), there is no blocking, so all four schedulers can get the full bandwidth aggregation effect. The STMS-C and STMS-A produce analogous results under the same buffer settings, which indicates the gap adjustment algorithm can track the true gap value precisely.

We then vary the latency of slow path. Both STMS-C and STMS-A improve the aggregation throughput over Default and ECF. We pick the improvement of STMS-A over Default as an example. fig. 14 shows the throughput of STMS-A normalized relative to that of Default. The improvement become more prominent as the paths become more heterogeneous and the buffer gets smaller.

Varying bandwidth Then we investigate the performance of our scheduler under network fluctuations. Here we change the bandwidth of both path randomly every 10 seconds. The bandwidth value is chosen from set {5, 10, 20, 30} Mbps. We generate 5 network configurations using different random seeds and run test five times for each network configurations.

fig. 15a shows the average throughput of four schedulers for each configuration. Note the error bar indicates the variability of the same configuration. Our scheduler outperforms other schedulers in every configuration. STMS-A performs even better than the STMS-C. This indicates that STMS-A is more adaptive to network fluctuations than the STMS-C.

Varying latency We simulate the varying latency condition using tc netem module. Both paths' latency is changed every 10 seconds, and the stddev of latency is set to 40% of the mean latency. We generate five latency configurations and run the test five times (fig. 15b). Similar to the bandwidth change scenario, STMS-A always performs best.

6 Evaluation in the wild

We next evaluate our scheduler in more realistic environments. The server is deployed in Aliyun [1] and has only one interface. The client is located inside our campus and connects to the server through WiFi and LTE. The China Telecom LTE cellular network incurs higher delay than the WiFi network. The average bandwidth and latency characteristic of each path are shown in table 2. We use tc to add extra latency on LTE path.

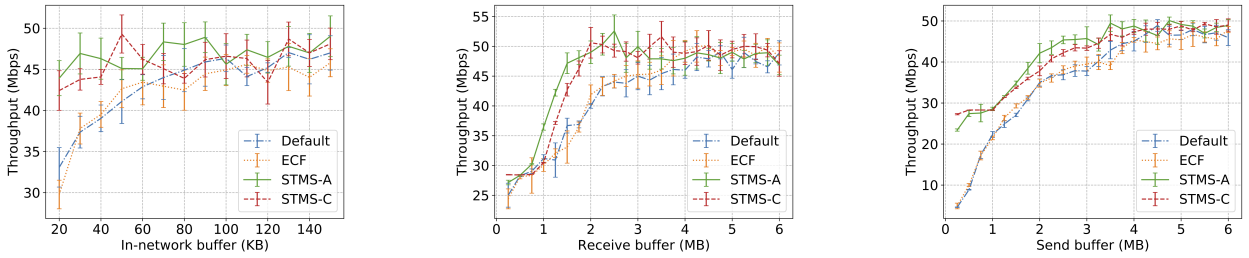
Table 2: Path characteristics

	Bandwidth(Mbps)	Latency(ms)
WiFi	40	50
LTE	30	70

We compare our scheduler against Default and ECF. We measure the download time of 200MB file of different schedulers. For each latency setting, we run the test five times. The result is shown in fig. 16. Both STMS-C and STMS-A outperform Default and ECF. The STMS-A can get the best performance, reducing the file download time by as much as 20% over Default.

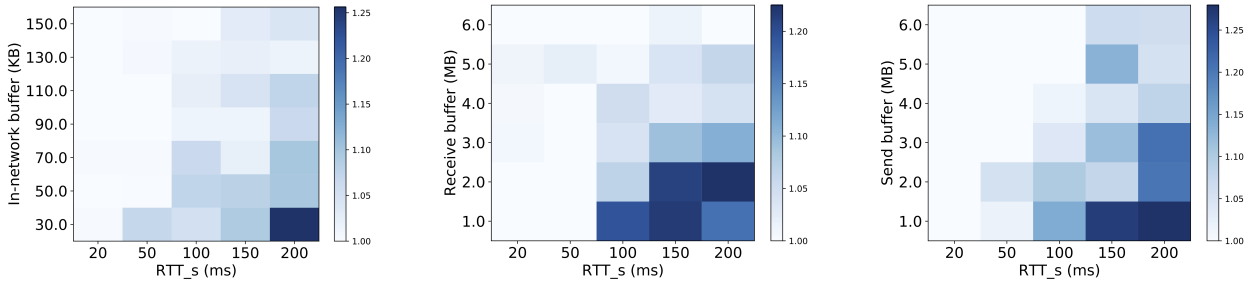
7 Related work

There are many studies on the improvement of MPTCP scheduler. To solve the host buffer problem, Raiciu *et al.* [26] propose the PR mechanism. Ferlin *et al.* [11] propose the Blocking Estimation-based Scheduler (BLEST) which aims to prevent the blocking by reducing the usage of slow path even if it has available CWND space. Both schedulers try to restrict the use of the slow path to alleviate the need of big host buffer resulting in the under-utilization of slow path.



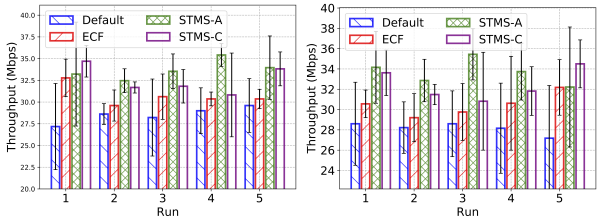
(a) Varying in-network buffer (b) Varying receive buffer (c) Varying send buffer

Figure 13: Throughput of different schedulers



(a) Varying in-network buffer (b) Varying receive buffer (c) Varying send buffer

Figure 14: STMS-A throughput normalized by Default



(a) Bandwidth change (b) Latency change

Figure 15: Throughput under dynamic network conditions

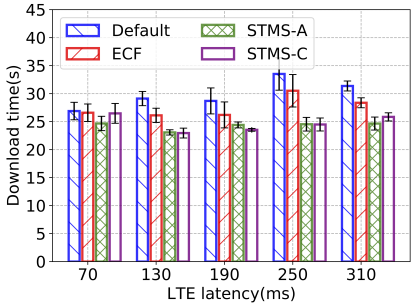


Figure 16: Average file download time in the wild (lower is better)

Kuhn *et al.* [22] propose DAPS to address the RTT difference of two paths. But it only considers the stable CWND and the scheduler running interval is very big thus can not react to the dynamic network changes.

Lim *et al.* [23] propose ECF which outperforms both DAPS and BLEST. But it only considers the tail packets.

Guo *et al.* [14] propose a new scheduler to balance two sub-flow completion time by sending packets inside a "chunk" in the opposite direction. Nonetheless, this approach will require a huge host buffer to store the whole chunk.

8 Conclusion

In this work, we identify a new root cause of MPTCP throughput degradation under heterogeneous path conditions. We propose STMS to effectively alleviate the problems due to the limitation in the host buffer size and in-network buffer size. Our experimental results show that STMS outperforms state-of-the-art schedulers in diverse network and buffer settings, especially when the path heterogeneity is large.

Acknowledgments. We thank the anonymous reviewers and our shepherd Theophilus Benson for their feedback on the paper. This work is supported by National Key R&D Program of China under Grant 2017YFB1010002, National 863 project (no. 2015AA015701) and a joint research with Protocol Research Lab, Huawei Technologies.

References

[1] Alibaba Cloud. https://www.alibabacloud.com/.

[2] AGACHE, A., DEACONESCU, R., AND RAICIU, C. Increasing datacenter network utilisation with grin. In *NSDI* (2015), pp. 29–42.

[3] APPENZELLER, G., KESLASSY, I., AND MCKEOWN, N. *Sizing router buffers*, vol. 34. ACM, 2004.

[4] APPLE Opens Multipath TCP In iOS11. http://www.tessares.net/highlights-from-advances-in-networking-part-1/.

[5] Balanced Linked Adaptation Congestion Control Algorithm for MPTCP. https://tools.ietf.org/html/draft-walid-mptcp-congestion-control-00.

[6] BRADEN, R. Requirements for internet hosts-communication layers.

[7] C. PAASCH, S. BARRE, E. A. Multipath TCP in the Linux Kernel. http://www.multipath-tcp.org.

[8] CARDWELL, N., CHENG, Y., GUNN, C. S., YEGANEH, S. H., AND JACOBSON, V. Bbr: Congestion-based congestion control. *Queue 14*, 5 (2016), 50.

[9] DENG, S., NETRAVALI, R., SIVARAMAN, A., AND BALAKRISHNAN, H. Wifi, lte, or both?: Measuring multi-homed wireless internet performance. In *Proceedings of the 2014 Conference on Internet Measurement Conference* (2014), ACM, pp. 181–194.

[10] ECF implementation in old MPTCP version. http://www.cs.umass.edu/~ylim/mptcp_ecf.

[11] FERLIN, S., ALAY, Ö., MEHANI, O., AND BORELI, R. Blest: Blocking estimation-based mptcp scheduler for heterogeneous networks. In *IFIP Networking Conference (IFIP Networking) and Workshops, 2016* (2016), IEEE, pp. 431–439.

[12] FORD, A., RAICIU, C., HANDLEY, M., AND BONAVENTURE, O. Tcp extensions for multipath operation with multiple addresses. Tech. rep., 2013.

[13] manpage of linux tc-fq . https://www.systutorials.com/docs/linux/man/8-tc-fq/.

[14] GUO, Y. E., NIKRAVESH, A., MAO, Z. M., QIAN, F., AND SEN, S. Accelerating multipath transport through balanced subflow completion. In *Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking* (2017), ACM, pp. 141–153.

[15] HAN, B., QIAN, F., JI, L., GOPALAKRISHNAN, V., AND BEDMINSTER, N. Mp-dash: Adaptive video streaming over preference-aware multipath. In *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies* (2016), ACM, pp. 129–143.

[16] HONDA, M., NISHIDA, Y., RAICIU, C., GREENHALGH, A., HANDLEY, M., AND TOKUDA, H. Is it still possible to extend tcp? In *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference* (2011), ACM, pp. 181–194.

[17] HUANG, J., QIAN, F., GERBER, A., MAO, Z. M., SEN, S., AND SPATSCHECK, O. A close examination of performance and power characteristics of 4g lte networks. In *Proceedings of the 10th international conference on Mobile systems, applications, and services* (2012), ACM, pp. 225–238.

[18] iPerf - The ultimate speed test tool for TCP, UDP and SCTP. https://iperf.fr/iperf-download.php.

[19] JACOBSON, V. Congestion avoidance and control. In *ACM SIGCOMM computer communication review* (1988), vol. 18, ACM, pp. 314–329.

[20] JIANG, H., WANG, Y., LEE, K., AND RHEE, I. Tackling bufferbloat in 3g/4g networks. In *Proceedings of the 2012 ACM conference on Internet measurement conference* (2012), ACM, pp. 329–342.

[21] KHALILI, R., GAST, N., POPOVIC, M., UPADHYAY, U., AND LE BOUDEC, J.-Y. Mptcp is not pareto-optimal: performance issues and a possible solution. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies* (2012), ACM, pp. 1–12.

[22] KUHN, N., LOCHIN, E., MIFDAOUI, A., SARWAR, G., MEHANI, O., AND BORELI, R. Daps: Intelligent delay-aware packet scheduling for multipath transport. In *Communications (ICC), 2014 IEEE International Conference on* (2014), IEEE, pp. 1222–1227.

[23] LIM, Y.-S., NAHUM, E. M., TOWSLEY, D., AND GIBBENS, R. J. Ecf: An mptcp path scheduler to manage heterogeneous paths. In *Proceedings of the 2017 ACM SIGMETRICS/International Conference on Measurement and Modeling of Computer Systems* (2017), ACM, pp. 33–34.

[24] NIKRAVESH, A., GUO, Y., QIAN, F., MAO, Z. M., AND SEN, S. An in-depth understanding of multipath tcp on mobile devices: measurement and system design. In *Proceedings of the 22nd Annual International Conference on Mobile Computing and Networking* (2016), ACM, pp. 189–201.

[25] PAASCH, C., KHALILI, R., AND BONAVENTURE, O. On the benefits of applying experimental design to improve multipath tcp. In *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies* (2013), ACM, pp. 393–398.

[26] RAICIU, C., PAASCH, C., BARRE, S., FORD, A., HONDA, M., DUCHENE, F., BONAVENTURE, O., AND HANDLEY, M. How hard can it be? designing and implementing a deployable multipath tcp. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation* (2012), USENIX Association, pp. 29–29.

[27] In Korean, Multipath TCP is pronounced GIGA Path. http://blog.multipath-tcp.org/blog/html/2015/07/24/korea.html.

[28] Why is the Multipath TCP scheduler so important ? http://blog.multipath-tcp.org/blog/html/2014/03/30/why_is_the_multipath_tcp_scheduler_so_important.html.

[29] SOMMERS, J., AND BARFORD, P. Cell vs. wifi: on the performance of metro area mobile connections. In *Proceedings of the 2012 ACM conference on Internet measurement conference* (2012), ACM, pp. 301–314.

[30] tc: Linux Advanced Routing and Traffic Control. http://lartc.org/lartc.html.

[31] WISCHIK, D., RAICIU, C., GREENHALGH, A., AND HANDLEY, M. Design, implementation and evaluation of congestion control for multipath tcp. In *NSDI* (2011), vol. 11, pp. 8–8.