Coupled Slow-Start: Improving the Efficiency and Friendliness of MPTCP's Slow-Start

Yansen Wang[1], Kaiping Xue[1*], Hao Yue[2], Jiangping Han[1], Qing Xu[3], Peilin Hong[1]
[1] Department of EEIS, University of Science and Technology of China, Hefei, Anhui 230027 China
[2] Department of Computer Science, San Francisco State University, San Francisco, CA 94132, USA
[3] Huawei Shanghai Research Institute, Shanghai 201206, China
[*] kpxue@ustc.edu.cn

***Abstract*—Multipath TCP (MPTCP) is designed to offer higher throughput than single-path TCP, and meanwhile MPTCP flow is fair to concurrent TCP flows at the bottleneck. Although the coupled congestion control in current MPTCP can achieve the goals by coupling different subflows, it only focuses on Congestion Avoidance but each subflow still behaves like an independent TCP flow in Slow-Start. However, during Slow-Start, MPTCP is unfair to concurrent TCP flows as it uses more network resources at the shared bottleneck than single-path TCP. Worse still, since the exponential growth of multiple concurrent subflows' congestion windows often results in serious buffer overflow and packet loss at the shared bottleneck, the latency of short flows using MPTCP is often not as good as using TCP. This leads to the fact that MPTCP cannot satisfy the above design goals when handling short flows. To address this issue, we present a Coupled Slow-Start (CSS) Algorithm in this paper. CSS links the exponential growth of subflows' congestion windows to ensure the fairness and reduce the burstiness of Slow-Start. To reduce the packet loss, CSS resets the Slow-Start Threshold (*ssthresh*) of different subflows for MPTCP to safely move to Congestion Avoidance when it achieves its *expected* throughput. Our simulation shows that CSS can lower short flows' latency of up to 45% and significantly reduce the packet loss in two typical network environments, meanwhile CSS is TCP-friendly at the shared bottleneck. Simulation results also indicate that CSS can perform at least as well as original MPTCP for the bulk data transfer in common network environments.**

***Index Terms*—MPTCP, Slow-Start, buffer overflow, friendliness**

I. Introduction

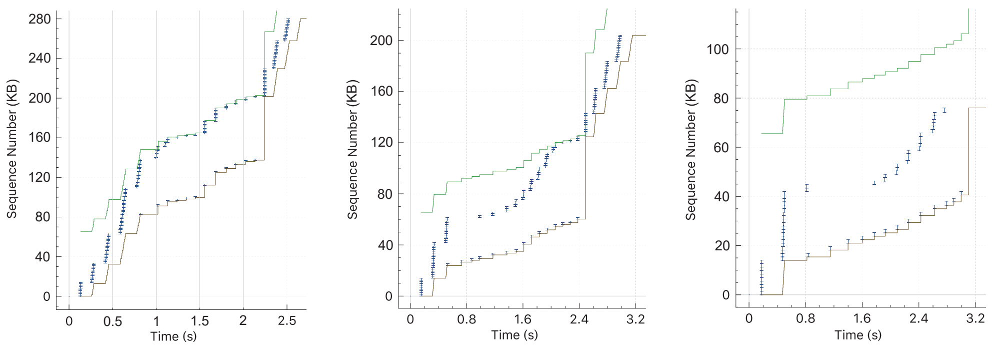
Multipath TCP (MPTCP) is an emerging technique to support the concurrent transmission by splitting its traffic across multiple available paths between end-hosts into multiple subflows [1]. Since modern computing devices are often equipped with multiple communication interfaces [2], such as mobile terminals with both Wi-Fi and cellular interfaces, servers with several high-speed interfaces in data centers, the concurrent use of multiple paths between multihomed end-hosts would efficiently utilize network's resources and significantly increase the resilience of the connectivity.

According to [3], MPTCP should satisfy the following design goals: (1) Improve throughput: perform at least as well as TCP. (2) Do no harm: take up no more capability than regular TCP at the shared bottleneck. Till now, a variety of coupled algorithms [3], [4] have been proposed to achieve these two goals. These algorithms couple the additive increase factor of each subflow's congestion windows (*cwnd*), which is designed not to increase more aggressively than single-path TCP. However, they only focus on Congestion Avoidance and each subflow still behaves like an independent TCP flow during Slow-Start. Recently, LISA [5] shows that MPTCP's unfriendly Slow-Start leads to heavy packet loss at the shared bottleneck. It proposes to subtract one existing subflow' *cwnd* by the Initial Window (IW) of the newly added subflow. However, the heavy packet loss is mainly caused by the unlimited exponential growth of concurrent subflows' *cwnd* and LISA fails to solve it actually.

In this paper, we identify performance issues in the current MPTCP's Slow-Start when its subflows share the same bottleneck. We find that MPTCP is unfair to concurrent TCP flows. For example, an MPTCP connection with two subflows would use twice as many resources (e.g., bandwidth and buffer) as single-path TCP does during Slow-Start. Unfairness becomes even severe as the IW and the number of subflows increase. Besides, the exponential growth of multiple concurrent subflows' *cwnd* will release large bursts of packets into network, which often overruns the buffer at the shared bottleneck. Worse still, since the packet loss at one subflow cannot limit the transmission of other subflows, MPTCP often fails to react to congestion timely. This further deteriorates the buffer overflow and causes severe packet loss. Given the fact that most flows across today's Internet are delay sensitive short flows [6], it is usually time-costly for short flows to recover from the serious packet loss. Therefore, the latency of short flows with MPTCP is longer that of TCP.

The main contributions are summarized as follows:

- We identify a common problem in the MPTCP's Slow-Start phase when its subflows share the same bottleneck. We observe that for short flows, MPTCP cannot satisfy the two design goals, i.e., it takes MPTCP more time to complete transmission despite more resources it uses when compared with TCP. The serious buffer overflow caused by MPTCP also leads to throughput deterioration of concurrent TCP flows.
- We propose a Coupled Slow-Start (CSS) Algorithm for MPTCP to mitigate the above problem. CSS slows down the exponential growth of subflows' *cwnd* when a new



(a) TCP (b) Subflow 1 of MPTCP (c) Subflow 2 of MPTCP

Fig. 1. Comparison: a 280KB transfer based on TCP and MPTCP

subflow joins in the connection to ensure the fairness and reduce the burstiness of Slow-Start. We also design an aggressiveness factor in CSS to serve the fairness and efficiency goals of MPTCP. When MPTCP achieves its *expected* throughput which is derived from the aggressiveness factor, CSS resets subflows' Slow-Start Threshold (*ssthresh*) to safely move to Congestion Avoidance without incurring risk of serious packet loss.

- We conduct extensive simulation and show that CSS can bring a goodput gain of up to 40% for short flows while improving performance of concurrent TCP flows. We also show that CSS does not degrade performance of long-lived MPTCP flows in common network environments.

The rest of this paper is organized as follows. In section II, we illustrate the performance issues in current MPTCP and analyze the reasons. Our proposed CSS is described in details in section III. Subsequently, we give performance evaluation in section IV and conclude our work in section V.

II. PROBLEM STATEMENT

After an MPTCP connection is established through 3-way handshake, the path manager of MPTCP will use the additional IP addresses advertised by the remote end-host to start new sublows in parallel [1]. In current MPTCP, subflows are uncoupled in Slow-Start and each behaves like an independent TCP flow. When a subflow receives an ACK, its *cwnd* is increased by 1 segment, doubling the window size per Round Trip Time (RTT) until it reaches Slow-Start Threshold (*ssthresh*) [3].

Because of the bursty transmission of packets in Slow-Start, the bottleneck router can be temporarily congested although the average sending rate within an RTT is much smaller than the bottleneck bandwidth. After new subflows join in the connection, the bursty traffic from multiple subflows often overruns the buffer at the shared bottleneck and causes serious packet loss. We observe that it takes MPTCP more time to complete a short transfer.

We illustrate this problem with an example of the Shared bottleneck scenario shown in Fig. 2(a). We use TCP and MPTCP to transfer a file of 200 packets (280KB). Fig. 1 shows the sequence number and ACK number progress versus time at the sender side of TCP and the two subflows of MPTCP. Each small horizonal line segment represents a segment sent at that time. There are two lines that bound the small horizonal line segments: the bottom one indicates the last acknowledged segment number (denoted as *snd_una*), and the top one indicates *snd_una* plus client's send window size (the send windows size is constant in simulation). In Fig. 1(c), subflow 2 of MPTCP starts about 0.25s later than subflow 1 and its time axis is 0.25s later than that of subflow 1. We can see clearly that TCP completes about 0.5s earlier than MPTCP. As shown in Fig. 1(a), TCP retransmits 12 packets in total. In Fig. 1(b) and Fig. 1(c), after subflow 2 joins, large bursts of traffic from the two subflows overrun the buffer at the shared bottleneck. Subflow 1 retransmits 15 packets and subflow 2 retransmits 13 packets in total. Recovering from the packet loss further delays transmission of MPTCP.

III. COUPLED SLOW-START ALGORITHM

To address the above issues, we present the CSS Algorithm for MPTCP, which only focuses on initial Slow-Start phase.

CSS includes two parts: 1) Coupled ssthresh and 2) Linked Smooth Growth. The former one uses an aggressiveness factor to serve the fairness and efficiency goals of MPTCP. It resets subflows' *ssthresh* for MPTCP to safely move to Congestion Avoidance, when it achieves its *expected* throughput which is derived from the aggressiveness factor. The latter one slows down the exponential growth of subflows' *cwnd* when a new subflow joins in the connection.

A. Coupled ssthresh

In TCP Reno's Slow-Start, the exponential growth of *cwnd* allows the sender to effectively discover available bandwidth. The *ssthresh* is the upper bound of *cwnd* to prevent the sender from congesting network with large bursts of packets. Unfortunately, if MPTCP's subflows share the same bottleneck, *ssthresh* often fails to timely limit the exponential growth of subflows' *cwnd* during Slow-Start. The reason is that *ssthresh* is the upper bound of each subflow' *cwnd*, then the upper bound of total *cwnd* is $n * ssthresh$ (n is the number of subflows), which is usually larger than the available capacity of the shared bottleneck. Without the limitation on *cwnd*, only preventing the MPTCP sender from congesting network with large bursts of packets based on the events of packet loss is impracticable, since the packet loss events detected at one subflow will not limit the transmission of other subflows.

To address this problem, we design an new aggressiveness factor, *con_ssthresh*, to limit the total *cwnd* of an MPTCP connection during Slow-Start. When the MPTCP connection achieves its *expected* throughput, we reset its subflows' *ssthresh* to exit Slow-Start and move to Congestion Avoidance. Its *expected* throughput depends on the *con_ssthresh*. To satisfy the fairness goals of MPTCP, we set *con_ssthresh* to be equal to TCP's *ssthresh* (the default value of *ssthresh* is set to a finite value (e.g., 65535)), aiming to only give the MPTCP connection as much throughtput as that in TCP when exiting Slow-Start. This idea is similar to the Linked Increase Algorithm (LIA) of MPTCP [3].

Based on *con_ssthresh*, we present the algorithm of Coupled ssthresh. It is illustrated in Algorithm 1 and executed when one of subflows receives a new ACK. Our algorithm first calculates the *current* sending rate $current = \sum_i \frac{subflow[i].cwnd}{subflow[i].RTT}$. Then, it defines $baseRTT$ to be the minimum of all subflows' current measured RTT. The expected throughput is calculated by $expected = con\_ssthresh/baseRTT$, where $con\_ssthresh$ is initialized to be the same default value of $ssthresh$. It is worth noting that if no packet loss occurs during Slow-Start, the *expected* throughput is that TCP would get on the MPTCP's best sub-paths. After that, our algorithm compares $current$ to $expected$. If $current > expected$, we reset subflows' *ssthresh* to exit Slow-Start and move to Congestion Avoidance. In this way, the *con_ssthresh* can prevent MPTCP from achieving additional throughput, which often results in packet loss at the shared bottleneck.

B. Linked Smooth Growth

Although our Coupled ssthresh algorithm above ensures that MPTCP and TCP achieve the same throughput when exiting Slow-Start without packet loss, unfairness still exists during Slow-Start. For example, when a new subflow joins, the total *cwnd* of MPTCP would have an increase of the IW (10 segments) of the new subflow. With a larger total *cwnd* when compared with TCP, MPTCP would use more resources at the bottleneck than TCP does. It will also shorten MPTCP's time to exit Slow-Start. Unfairness becomes more severe as the IW and number of subflows increase.

The solution of Linked Smooth Growth is to slow down the growth of exiting subflows' *cwnd* when a new subflow joins, aiming to ensure that it takes the MPTCP connection as much time as a simultaneously started TCP flow to exit Slow-Start if no loss occurs. This approach can alleviate the unfairness between MPTCP and TCP, while reducing the burstiness and packet loss of Slow-Start [7].

We present the algorithm of Linked Smooth Growth. It is illustrated in Algorithm 2 and executed when a new subflow joins. Define α to be the growth factor of subflows' *cwnd*, i.e., the ratio of *cwnd* between two sequential RTTs. α is 1.5 as the receiver usually uses delayed ACK [8]. The algorithm first estimates the time (in RTT) needed for the MPTCP connection to reach its *con_ssthresh* from current total *cwnd*. When a new subflow joins, the total *cwnd* increases by IW. Then, the new growth factor, which would be lower than before, is calculated

Algorithm 1: Coupled ssthresh

```
1  Initialize initialSS = true, con_ssthresh = ssthresh
   // the connection is in initial Slow-Start
2  for each subflow $i$ do
       // subflows are in initial Slow-Start
3      subflow[i].initialSS = true
4  end
5  if subflow $i$ experiences a loss then
6      subflow[i].initialSS = false
7  end
8  if subflow $i$ receives a new ACK then
9      baseRTT = get_current_minRTT()
10     if $\sum_j \frac{subflow[j].cwnd}{subflow[j].RTT} > \frac{con\_ssthresh}{baseRTT}$ and initialSS
       == true then
11         initialSS = false
12         for each subflow $j$ do
13             if subflow[j].initialSS == true then
14                 subflow[j].ssthresh = subflow[j].cwnd
                   subflow[j].initialSS = false
15             end
16         end
17     else if subflow[i].cwnd > ssthresh and
       subflow[i].initialSS == true then
           // enter CA phase separately
18         subflow[i].initialSS = false;
19     end
20 end
```

Algorithm 2: Linked Smooth Growth

```
1  Initialize $\alpha = 1.5$, con_ssthresh = ssthresh
2  if a new subflow joins then
3      baseRTT = get_current_minRTT()
4      totalcwnd = $\sum_i subflow[i].cwnd$
5      newTotalcwnd = totalcwnd + IW
6      RoundTrip = $\log_\alpha \frac{con\_ssthresh}{totalcwnd}$
7      $\alpha = \sqrt[RoundTrip]{\frac{con\_ssthresh}{newTotalcwnd}}$
8      for each subflow $i$ do
9          subflow[i].$\alpha$ = $\alpha^{\frac{subflow[i].RTT}{baseRTT}}$;
10     end
11 end
```

in line 7. However, if the RTT in the new subflow's path is larger than that of the exiting subflows, it would take more time to double its *cwnd*. Sometimes, the total *cwnd* of an MPTCP connection would even be smaller than the *cwnd* of a simultaneously started TCP flow. To avoid this, line 9 compensates the growth factor of subflows with large RTT. In Algorithm 3, we use Byte Counting algorithm [8] to accurately control the growth of subflows' *cwnd*. In line 3, we use α calculated in Algorithm 2 to increase subflows' *cwnd* based on the acked bytes in each ACK. We also limits the maximum increase of *cwnd* for each ACK in the case that ACKs are dropped and acked bytes are too much.

Algorithm 3: Subflow Byte Counting

```
1 Initialize acked_Bytes
2 When increase subflow's cwnd for the ACK it receives:
  if subflow[i].initialSS == true then
3     subflow[i].cwnd += min(2*MSS, (subflow[i].$\alpha$ - 1) $*$ acked_Bytes
4 else
5     subflow[i].cwnd = subflow[i].cwnd + 1
6 end
```

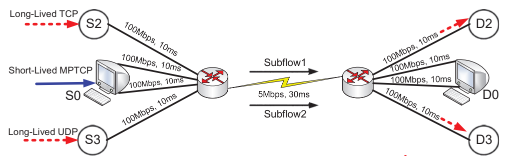
IV. Performance Evaluation

In this section, we evaluate our CSS using ns-3 simulator [9] and compare it with TCP, original MPTCP and LISA.

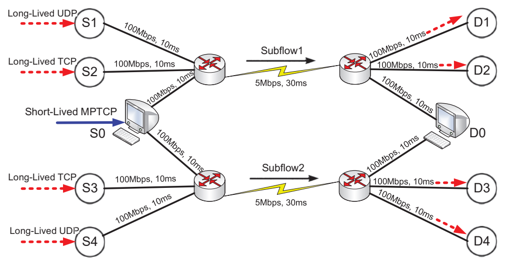
A. Simulation Setup

We consider two typical scenarios, Shared bottleneck scenario shown in Fig. 2(a) and Non-shared bottleneck scenario shown in Fig. 2(b).

1) **Shared bottleneck scenario**: in this scenario, the network has a common bottleneck. A short MPTCP flow with two subflows or a short TCP flow will start randomly in the interval [2 sec, 5 sec] from S_0 to D_0. A long-lived TCP background flow is transmitted from S_2 to D_2. A long-lived UDP background flow exists from S_3 to D_3 at constant rate of 500 Kbps. These two long-lived flows exist in the whole transfer of the short flow. The bottleneck bandwidth is set to 5 Mbps and the link delay is set to 30 ms. Bandwidth of other links is set to 100 Mbps and their link delay is set to 10 ms. We assume that RTT of the path is 100 ms and BDP (Bandwidth-Delay Product) of the path is about 47 packets (In our simulation, the packet size is set to 1400 bytes.). The Drop-Tail buffer size at the bottleneck is set to the BDP (47 packets) of the path.



(a) Shared Bottleneck Scenario



(b) Non-shared Bottleneck Scenario

Fig. 2. Topologies used for CSS evaluation

2) **Non-shared bottleneck scenario**: in this scenario, the network has two distinct bottlenecks. Also, a short MPTCP flow with two subflows or a short TCP flow will start randomly in the interval [2 sec, 5 sec] from S_0 to D_0. Each bottleneck has a long-lived TCP and UDP background flow. The long lived TCP flow exists from S_2 to D_2 and from S_3 to D_3, respectively. The long lived UDP flow exists from S_1 to D_1 and from S_4 to D_4 at constant rate of 500 Kbps, respectively. Parameters of bottlenecks and other links are the same as in Shared bottleneck scenario.

We will change the bottleneck buffer size to evaluate the adaptability of our CSS in various network environments, since the bottleneck buffer size can significantly affect the performance of short flows.

We consider the following metrics for the short flow:

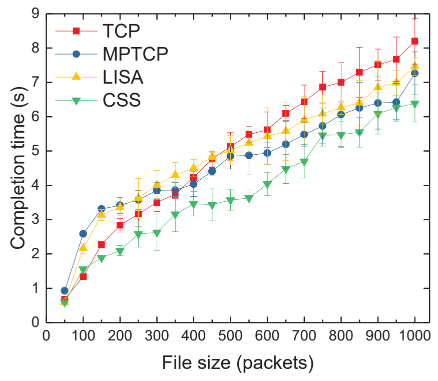
- Completion time, which is the time from sending the first SYN packet to receive the ACK for the last packet in a file transfer.
- Retransmissions, which is the total number of retransmitted packets during the transfer.
- *Sum of cwnd*, which is the total *cwnd* of an MPTCP connection when it completes the file transfer. For clarity, it is also denoted as the *cwnd* of a TCP connection.

In simulation, TCP's *ssthresh* is set to 65535 and *con_ssthresh* is equal to *ssthresh*. We run each experiment 10 times and present the average result and its standard deviation.

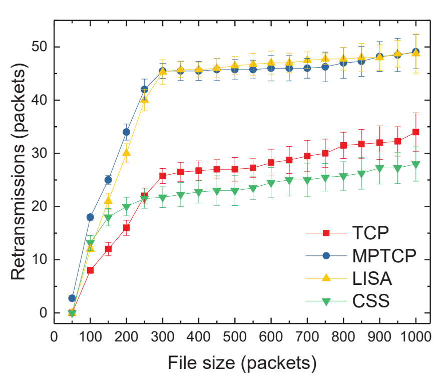
B. Shared Bottleneck Scenario

Fig. 3(a) shows the average file completion time and Fig. 3(b) shows the corresponding retransmissions of TCP, original MPTCP, LISA and CSS, with file size varying from 50 packets (70KB) to 1000 packets (1.4MB). The bottleneck buffer size is set to the BDP (47 packets) of the path. We observe that TCP outperforms MPTCP and LISA in completion time when the file size is less than 400 packets and the reason is that MPTCP and LISA have more retransmissions. In Fig. 3(b), CSS significantly reduces MPTCP's packet loss by half. As a result, CSS outperforms the other schemes and reduces the completion time of MPTCP by about 0.5s on average and up to 1.5s in the best case. Packet loss is caused by the bursty traffic during Slow-Start, so the retransmissions will not increase rapidly when file size exceeds 300 packets as the Slow-Start has been terminated. The packet loss of CSS and TCP is similar because MPTCP's *con_ssthresh* is equal to TCP's *ssthresh*, indicating that their aggressiveness is also similar. LISA does not limit the exponential growth of its subflows' *cwnd*, therefore LISA and MPTCP often have nearly the same results.

To evaluate the adaptability of CSS in various network environments, we vary the bottleneck buffer size from 20 to 130 with the file size of 400 packets (560KB). Fig. 5(a) and Fig. 5(b) show the average file completion time and the corresponding retransmissions, respectively. With buffer size increasing, all schemes' retransmissions decrease, so their completion time also decreases. This is because the congestion at the bottleneck can be absorbed by queuing bursty packets in the large buffer. When buffer size exceeds 30 packets, CSS can drastically reduce packet loss, and continuously bring a goodput gain over MPTCP. However, when the buffer size exceeds 70 packets, CSS and TCP experience no packet loss,



(a) Mean completion time vs file size



(b) Retransmissions vs file size

Fig. 3. Shared bottleneck with file size varying and buffer size fixed at 45 packets

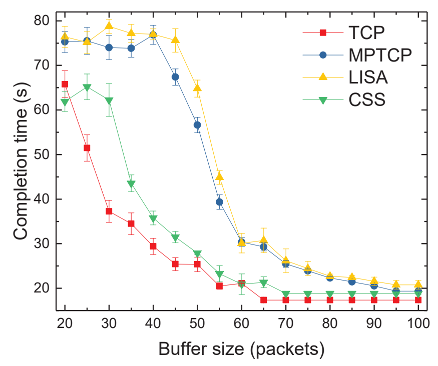
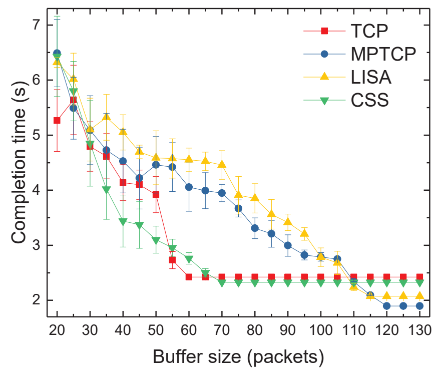
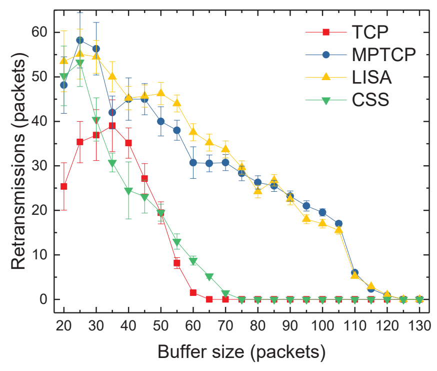


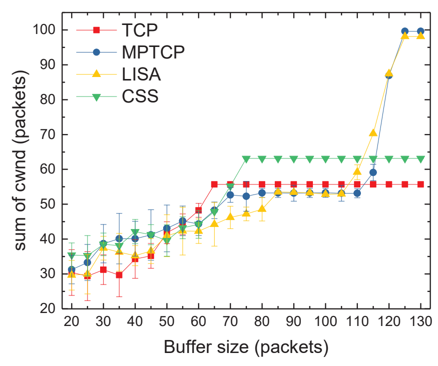
Fig. 4. Long-lived TCP flow's completion time vs buffer size



(a) Mean completion time vs buffer size



(b) Retransmissions vs buffer size



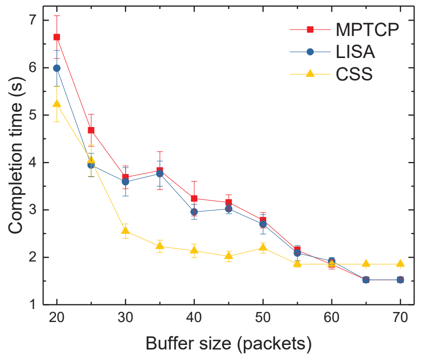
(c) *Sum of cwnd* when completing transfer

Fig. 5. Shared bottleneck with buffer size varying and file size fixed at 400 (5.6KB) packets

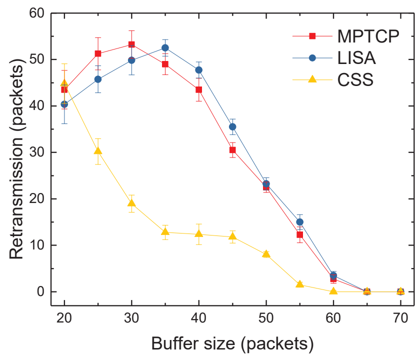
therefore their completion time reaches their minimums. These minimums are caused by their upper bounds on total *cwnd*, i.e., the *ssthresh* and *con_ssthresh*. It is also the same reason for the completion time of MPTCP and LISA. We will discuss it in detail below.

To avoid buffer overflow, CSS resets subflows' *ssthresh* to move to Congestion Avoidance. However, this may result in that MPTCP exit Slow-Start with a small *cwnd* and it will harm the performance of bulk data transfer as compared with original MPTCP. To evaluate this tradeoff, Fig. 5(c) plots the *Sum of cwnd* when completing the 400 packets (560KB) file transfer as a function of bottleneck buffer size. Note that after transmitting 400 packets, all schemes have been in Congestion Avoidance for a period of time regardless of the buffer size. So a larger *Sum of cwnd* at that time indicates a better support for the subsequent bulk data transfer. When the buffer size is less than 115 packets, the *Sum of cwnd* of CSS is greater than (or similar to) that of MPTCP and LISA, due to the fact that packet loss has terminated MPTCP and LISA's Slow-Start. When buffer size exceeds 115 packets, the *Sum of cwnd* of MPTCP and LISA outperforms that of CSS and TCP. The reason is that MPTCP and LISA experience no packet loss during Slow-Start, so their total *cwnd* reach the upper bound, i.e., *2*ssthresh*. Since CSS and TCP use *con_ssthresh* and *ssthresh*, respectively, to limit their total *cwnd*, it takes a little longer to complete the transfer as shown in Fig. 5(a)). However, recent research [10] suggests that buffers in the backbone routers can be significant small. Besides, considering the Random Early Detection (RED) algorithm deployed in routers, no packet loss occurs during Slow-Start rarely happens. Thus, we claim that CSS supports the bulk data transfer at least as well as original MPTCP in common network environments.

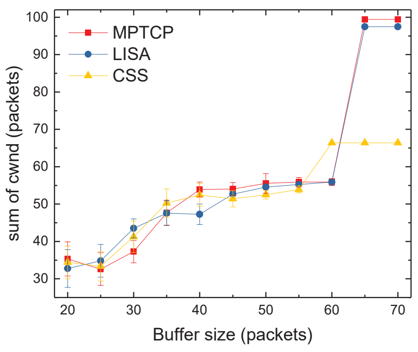
In order to compare the aggressiveness of different MPTCP Slow-Start schemes with TCP's Slow-Start, we evaluate the impact of different MPTCP's Slow-Start schemes and TCP's Slow-Start on the completion time of a concurrent long-lived TCP flow. We use the long-lived TCP flow to transfer a file of 4000 packets (5.6MB). Meanwhile, we start 80 TCP or MPTCP short flows in sequence to ensure that the short flows exist during the lifetime of the long-lived TCP flow. Each short flow transfers a file of 200 packets (280KB). The intervals between two sequential start-ups are uniformly distributed and the average is 1.2s. Fig. 4 shows completion time of the long-lived TCP flow under different Slow-Start schemes, with bottleneck buffer size varying from 20 to 95 packets. We observe that the MPTCP and LISA's Slow-Start are more aggressive than TCP's Slow-Start. The completion time of the long-lived TCP flow when short flows use MPTCP and LISA is 265% the time when short flows use TCP at buffer size of 45 packets, due to more retransmissions they cause. We can conclude that CSS shows similar aggressiveness with TCP.



(a) Mean completion time vs buffer size

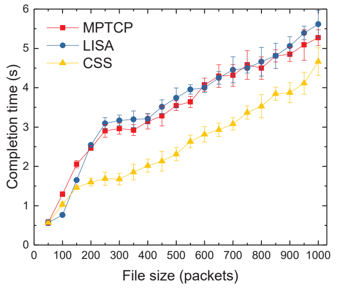


(b) Retransmissions vs buffer size

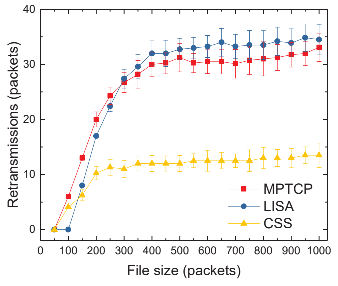


(c) *Sum of cwnd* when completing transfer

Fig. 6. Non-Shared bottleneck with buffer size varying and file size fixed at 400 (5.6KB) packets



(a) Mean completion time vs file size



(b) Retransmissions vs file size

Fig. 7. Non-Shared bottleneck with file size varying and buffer size fixed at 45 packets

C. Non-shared Bottleneck Scenario

We also first vary the file size from 50 packets (70KB) to 1000 packets (1.4MB) with bottleneck buffer size of the BDP (47 packets) of the path. Fig. 7(a) and Fig. 7(b) show the average completion time and the corresponding retransmissions, respectively. We observe that CSS reduces the completion time by about 1.2s on average as compared with MPTCP. As shown in Fig. 7(b), CSS reduces the packet loss by 50%-67% when the file size exceeds 100 packets.

To evaluate the adaptability of CSS in Non-shared bottleneck scenario, we vary the bottleneck buffer size from 20 to 70 packets with the file size of 400 packets (560KB). Fig. 6(a) shows the average completion time and Fig. 6(b) shows the corresponding retransmissions. When the buffer size is less than 65 packets, CSS can significantly reduce packet loss and bring a goodput gain of up to 45% to MPTCP. When buffer size exceeds 65 packets, MPTCP and LISA outperform CSS in completion time. The reason is the same as in Shared bottleneck scenario.

The *con_ssthresh* in CSS may unnecessarily limit subflows' *cwnd* and harm the performance for the bulk data transfer in Non-shared bottleneck scenario. To evaluate this tradeoff, Fig. 6(c) plots the *Sum of cwnd* when completing 400 packets (560KB) file transfer as a function of bottleneck buffer size. When the buffer size is less than 65 packets, the *Sum of cwnd* of CSS is similar to that of MPTCP and LISA. However, when the buffer size exceeds 65 packets, the buffer is large enough so that MPTCP and LISA experience no packet loss during Slow-Start. Therefore MPTCP and LISA outperform CSS, as discussed in Shared bottleneck scenario.

V. Conclusion

This paper has illustrated that when MPTCP's subflows share a common bottleneck, MPTCP's uncoupled Slow-Start results in that MPTCP cannot satisfy its fairness and efficiency goals for short flows. To address this issue, we present a Coupled Slow-Start algorithm for MPTCP. CSS links the exponential growth of subflows' *cwnd* and resets the *ssthresh* of different subflows when it achieves its expected throughput. Simulation shows that CSS can improve the latency of short flows and perform as well as original MPTCP for the bulk data transfer in common network environments.

VI. Acknowledgment

This work is supported by the National Natural Science Foundation of China under Grant No. 61379129 and No. 61671420, Youth Innovation Promotion Association CAS, and the Fundamental Research Funds for the Central Universities.

References

[1] A. Ford, C. Raiciu, M. Handley, O. Bonaventure, and C. Paasch, "TCP extensions for multipath operation with multiple addresses," *IETF RFC, RFC6824*, 2016.

[2] D. Ni, K. Xue, P. Hong, H. Zhang, and H. Lu, "OCPS: Offset compensation based packet scheduling mechanism for multipath TCP," in *Proceedings of 2016 IEEE International Conference on Communications (ICC 2016)*. IEEE, 2015, pp. 6187–6192.

[3] C. Raiciu, M. Handley, and D. Wischik, "Coupled congestion control for multipath transport protocols," *IETF RFC, RFC6356*, 2011.

[4] K. Xue, J. Han, H. Zhang, K. Chen, and P. Hong, "Migrating unfairness among subflows in MPTCP with network coding for wired–wireless networks," *IEEE Transactions on Vehicular Technology*, vol. 66, no. 1, pp. 798–809, 2017.

[5] R. Barik, M. Welzl, S. Ferlin, and O. Alay, "LISA: A linked Slow-Start algorithm for MPTCP," in *Proceedings of 2016 IEEE International Conference on Communications (ICC 2016)*, 2016, pp. 1–7.

[6] M. Kheirkhah, I. Wakeman, and G. Parisis, "Short vs. long flows: a battle that both can win," in *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4. ACM, 2015, pp. 349–350.

[7] H. Wang, H. Xin, D. S. Reeves, and K. G. Shin, "A simple refinement of Slow-Start of TCP congestion control," in *Proceedings of the 5th IEEE Symposium on Computers and Communications (ISCC 2000)*. IEEE, 2000, pp. 98–105.

[8] M. Allman, "TCP byte counting refinements," *ACM SIGCOMM Computer Communication Review*, vol. 29, no. 3, pp. 14–22, 1999.

[9] "MPTCP NS3 code," http://code.google.com/p/mptcp-ns3/.

[10] G. Appenzeller, I. Keslassy, and N. McKeown, "Sizing router buffers," *ACM SIGCOMM Computer Communication Review*, vol. 34, no. 4, pp. 281–292, 2004.