Reducing the Flow Completion Time for Multipath TCP

GeonYeong Heo[1] and Joon Yoo[2*]
[1 2] Department of Software, Gachon University
Seongnam, Korea
[e-mail: heo.geoyeo@gmail.com, joon.yoo@gachon.ac.kr]
*Corresponding author: Joon Yoo

Received May 29, 2018; revised November 10, 2018; revised January 31, 2019; accepted February 3, 2019; published August 31, 2019

Abstract

The modern mobile devices are typically equipped with multiple network interfaces, e.g., 4G LTE, Wi-Fi, Bluetooth, but the current implementation of TCP can support only a single path at the same time. The Multipath TCP (MPTCP) leverages the multipath feature and provides (i) robust connection by utilizing another interface if the current connection is lost and (ii) higher throughput than single path TCP by simultaneously leveraging multiple network paths. However, if the performance between the multiple paths are significantly diverse, the receiver may have to wait for packets from the slower path, causing reordering and buffering problems. To solve this problem, previous MPTCP schedulers mainly focused on predicting the latency of the path beforehand. Recent studies, however, have shown that the path latency varies by a large margin over time, thus the MPTCP scheduler may wrongly predict the path latency, causing performance degradation. In this paper, we propose a new MPTCP scheduler called, choose fastest subflow (CFS) scheduler to solve this problem. Rather than predicting the path latency, CFS utilizes the characteristics of these paths to reduce the overall flow completion time by redundantly sending the last part of the flow to both paths. We compare the performance through real testbed experiments that implements CFS. The experimental results on both synthetic packet generation and actual Web page requests, show that CFS consistently outperforms the previous proposals in all cases.

Keywords: Multipath TCP, Wireless Network, Flow completion time, MPTCP Scheduler

A preliminary Korean version of this paper was submitted as a Masters thesis at Gachon University. This work was supported by the Basic Science Research Program through the National Research Foundation of Korea supported by MSIT (NRF-2017R1D1A1B03033934) and by the Korean MSIT (Ministry of Science and ICT), under the National Program for Excellence in SW (2015-0-0093), supervised by the IITP (Institute for Information & communications Technology Promotion).

1. Introduction

In recent years, the production and demand of mobile devices e.g., smart phones, tablet PCs, that have multiple network interfaces, such as 4G LTE, Wi-Fi, and Bluetooth, have increased dramatically. This behavior renders an opportunity to utilize the multiple network interfaces to provide better robustness and efficiency. For example, Apple's iOS 7 supports multipath TCP (MPTCP), as defined in RFC 6824 [1]. Additionally, KT, South Korea's largest telecommunications company, has provisioned an MPTCP proxy server to allow users to leverage LTE and Wi-Fi simultaneously [2].

MPTCP is an extension of TCP and uses two or more paths, each as subflows, by extending the single TCP connection. MPTCP is similar to TCP in many ways, but it provides additional features to utilize the multiple network interfaces. For example, if one connection is lost, data communication can still continue via the remaining path, providing robustness to link failures. MPTCP can also provide higher throughput, because it can simultaneously leverage multiple paths. Furthermore, MPTCP provides backward compatibility; if one node does not support MPTCP, the other node can revert to using the legacy TCP connection.

Unlike the traditional TCP, the MPTCP packets are sent to multiple paths, which may have diverse latency, causing out-of-order delivery at the MPTCP receiver. Here, the receiver should store the unordered packets in its receive buffer, and reorder them for orderly reception at the application layer. If the paths have diverse delay performance, which is evidenced in recent measurement studies [3], the receiver buffering can be extreme, causing performance degradation. To overcome this issue, some MPTCP schedulers tried to predict the path performance beforehand, and redesign the MPTCP scheduler accordingly [4,5,6]. These approaches generally show better performance, but if the path latency fluctuates over time, they may suffer from the inaccurate predictions. Recent research has found that two thirds of the Internet latency samples exceeded 100ms of variation, supporting our argument [7].

In this paper, we propose a new MPTCP scheduler called choose fastest subflow (CFS) scheduler. CFS reduces the overall flow completion time by cleverly sending the last part of the flow to both paths. We give some simple numerical analysis to show the benefits of CFS [20-26]. We compare the performance through two types of real testbed experiments, namely synthetic packet generation and actual Web page requests, and show the CFS performance compared to the previous work.

This paper is organized as follows. In Section 2, we introduce the basic operation principles of MPTCP and it scheduler. In Section 3, we show related work on the MPTCP scheduler. In Section 4, we present our proposal, CFS, and provide algorithms and numerical analysis. Section 5 describes the experimental testbed setup and Section 6 describes the results of the experiments from the testbed. Finally, in Section 7, we conclude this paper.

2. Background

In this section, we discuss the basic operating principles and problems of MPTCP. First, we describe the basic concept and structure of MPTCP and its default scheduler. Due to the multipath transmission, the out-of-order problem can occur at the receiver's buffer.

2.1 MPTCP

MPTCP is a relatively new standard that enables nodes to simultaneously utilize various network interfaces. Unlike TCP, which can only transmit data over a single path, MPTCP uses multipath. Therefore, even if one connection is lost, it can still transmit data. It can also provide higher throughput, by simultaneous utilizing multipaths [8]. **Fig. 1** shows the basic structure of MPTCP [9]. It is recognized as TCP in the application layer, but when data comes to the transport layer through a socket, it is allocated to multiple paths on different interfaces. Setting up the MPTCP connection is similar to the original TCP three-way handshake: SYN, SYN/ACK, and ACK. The only difference is the MP_CAPABLE option in each packet, which contains a key value to add a new subflow [10]. After the first flow is established, another three-way handshake with the MP_JOIN option is used to add another MPTCP subflow.

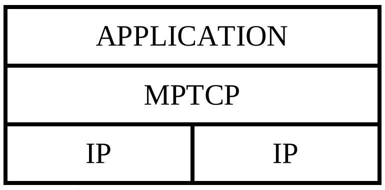


Fig. 1. MPTCP Structure

2.2 MPTCP Scheduler

Unlike the legacy TCP, MPTCP transmits data using multiple subflows, thus it needs to determine which path to use when transmitting each packet. The MPTCP scheduler is responsible for this decision [11]. The default MPTCP scheduler uses the minimal round-trip time (RTT) scheduler that finds a subflow with the lowest RTT. First, it allocates packets to the lowest RTT subflow until the congestion window (*cwnd*) becomes full, then finds the next fastest subflow, and allocates the remaining packets, and so on. **Fig. 2** shows the MPTCP scheduler as defined in the current implementation [12]. In this example, packets 1 to 3 are first scheduled to the first subflow with the lowest RTT, then the remaining ones are sent to the next subflow.

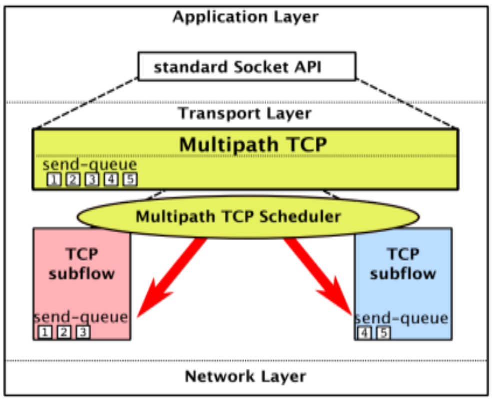


Fig. 2. MPTCP Scheduler [11]

2.3 “Out-of-Order” Problem

Since the packets sent in MPTCP may travel through multiple paths with diverse delay, there is a high probability that the packets will not arrive in order at the receiver. The out-of-order packets at the receiver have to be buffered at the receiver’s buffer space, since they will not be transferred to the application layer until the delayed packets arrive.

Let us give a simple example. We assume the sender transmits packets 1 to 30 to the receiver, as shown in **Fig. 3**, using the MPTCP default scheduler. Per the given minimal RTT (minRTT) scheduler, packets are transmitted first by the lowest RTT subflow (*subflow*$_f$) until the *cwnd* is full. For example, the first 10 packets are sent to subflow$_f$. Next, the minRTT scheduler transmits the remaining packets (11 to 20) to the next-fastest subflow (*subflow*$_s$). Since subflow$_f$ has a much smaller RTT (10ms vs. 100ms), it can transmit the remaining packets (21 to 30) using subflow$_f$, and they arrive at the receiver before packet 11 to 20. In summary, packets 1 to 10 and 21 to 30 arrive at the receiver, whereas packets 1 to 10 are already sent to the application layer. Because packets 11 to 20 have not arrived until 50ms, i.e., half of *subflow*$_s$ RTT 100ms, packets 21 to 30 continuously occupy the buffer space of the receiving side. In a severe situation where the two paths’ RTT are diverse, there is a chance that the receiver buffer becomes full, and the fast subflow is no longer utilized until the delayed packets from the slow subflow arrive. This situation is called *head-of-line blocking*.

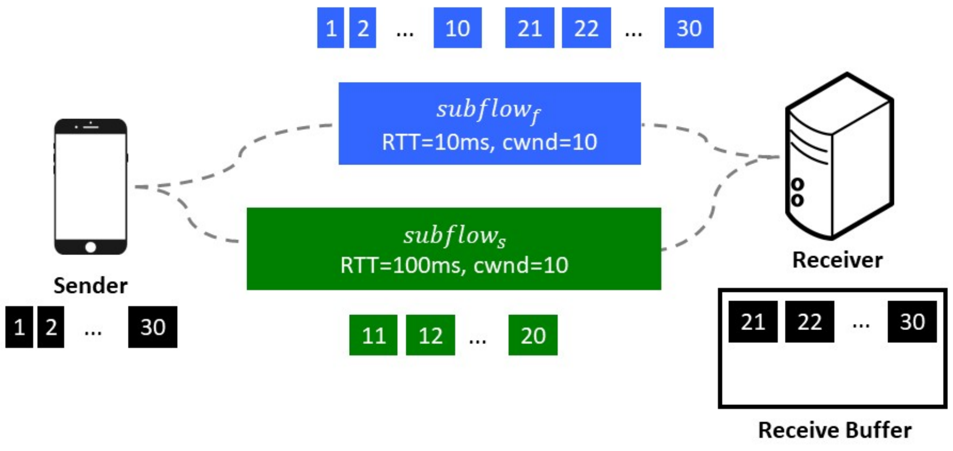


Fig. 3. Out-of-Order Problem

To solve this problem, there is an opportunistic method of retransmitting packets from the slow subflow to the faster subflow, called *opportunistic retransmission* [13]. When the receive window becomes 0 due to the out-of-order packets, the sender perceives that the receiver has caused head-of-line blocking and retransmits the packets that were in the slow subflow at the fast subflow, provided that the fast subflow’s *cwnd* is not full. However, the opportunistic retransmission is performed only when the buffer on the receiving side is full. When the flow size is short, the problem persists since the buffer does not become full. Thus, opportunistic retransmission is not activated.

Next, there is a penalization (PR) method that reduces the *cwnd* of the slow subflow by half to prevent it from being further utilized [13]. However, today's routers and APs have expanded their buffer sizes to prevent packets from being lost during transmission. Thus, small buffered data, including ACKs, may be buffered, causing buffer bloat [11]. Therefore, in the case of PR, it is difficult to restore the reduced *cwnd* to its original size.

If the performance differences (e.g., RTT, bandwidth) between two paths is small, and the MPTCP is used more than a single TCP, higher throughput can be expected. Thus, large-flow (i.e., large files) can benefit, but short-flow (i.e., smaller files, message services) are more affected by delay than bandwidth. Therefore, MPTCP is more sensitive to delays than high bandwidth, in the case of the small files. Therefore, by overcoming the performance differences between the two paths, it is possible to improve performance.

3. Related Work

In this section, we discuss some recently proposed MPTCP schedulers that attempt to solve the aforementioned out-of-order problem.

Delay-Aware Packet Scheduler (DAPS) [4] is a scheduler for solving head-of-line blocking and out-of-order problems when using multiple paths. The next packet to be sent is assigned to a subflow based on the previous *cwnd* and RTT, so that the packets sent will arrive in order. As the RTT increases, less packets are allocated, and, as the RTT decreases, more packets are allocated. However, because RTT varies with time [7], if its variation is large over time, the DAPS scheduler may incorrectly distribute packets.

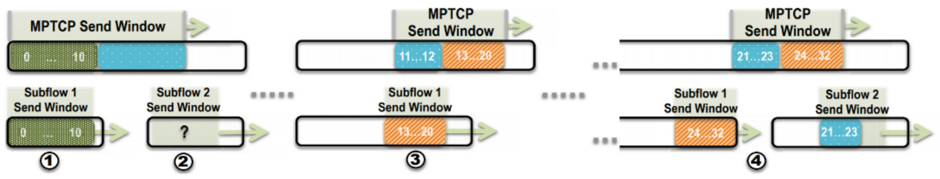


Fig. 4. BLEST Scheduling [5]

Blocking Estimation-Based MPTCP Scheduler (BLEST) [5] is a scheduler that enables packets to arrive in order based on the size of the previous transmission's send window. The minRTT scheduler sends its *cwnd*, in order, from the faster subflow to the slower subflow. However, if the send widow is filled by the window of a slower subflow, a faster subflow will not be able to use it. Therefore, BLEST reduces the chance of sending to the slower subflow; it sends the data by allocating as much as the *cwnd* through the fast subflow's ACK. For example, in **Fig. 4**, subflow 2 has a higher RTT than subflow 1. In the case of the minRTT scheduler, the packets are transmitted first to subflow 1's as much as the *cwnd* size and then to the slower subflow 2. If the minRTT scheduler is used as-is, the slower subflow 2 occupies the current send window, and the faster subflow 1 cannot transmit data. To prevent this, BLEST's scheduler allocates as much as the slow subflow's *cwnd*. However, when the fast subflow can receive and send ACK, it transmits data only using fast subflow.

The ECF scheduler [6] aims to reduce the completion time of short flows. If there is a large performance difference between the two paths, but the slow subflow is still utilized, then the fast subflow must transmit all of its packets and then wait for the transmission of the packets sent by the slow subflow to be completed. The key idea here is as follows; each subflow's *cwnd*, the RTT, and the k (i.e., byte unit) remaining in the connection level's send buffer prior to packet allocation determines whether or not to use the slow subflow. If using a slower subflow will complete sooner than using a faster subflow, it uses the slower subflow. If using a slower subflow takes more time than the fast subflow, rather than using the slow subflow, it waits for the fast subflow to finish its current transmission, and then send the rest of the traffic to the fast subflow. Like BLEST, the ECF determines whether to use a slower subflow or not. However, the ECF focuses on the packets' arrival time, not order. For BLEST, the choice is based on the send window. For ECF, the choice is based upon the connection level send buffer, the *cwnd* of each subflow, and the RTT.

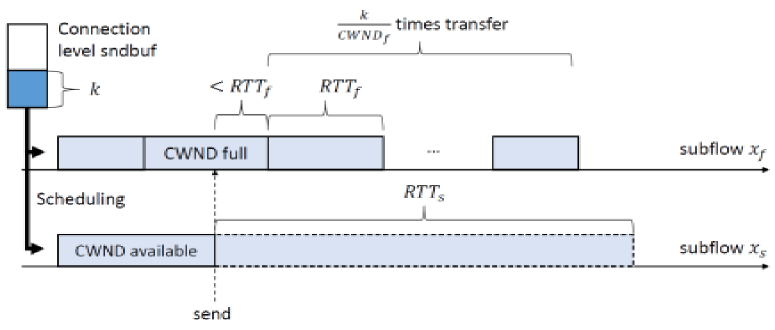


Fig. 5. ECF Scheduling [6]

Fig. 5 shows the basic mechanism of ECF. If the slow subflow takes more time than the fast subflow, ECF waits to use the fast subflow. Thus, ECF is the MPTCP scheduler that shows the best performance when the difference between two paths is large. This method can improve MPTCP performance, because it can correctly predict network fluctuations in RTT and *cwnd*, under stable conditions. The work in [6] show that ECF outperforms both DAPS and BLEST. However, when the performance fluctuates over time, the MPTCP scheduler may wrongly predict the path performance in advance. Hence, the performance enhancement diminishes.

In Freeze MTCP [14], the scheduler freezes the slow path if the RTT difference is above a certain threshold, e.g., 100ms. Freeze MPTCP improves the default MPTCP by a large margin. The key idea is simple, but it did not show any comparisons with the previous approaches.

3. Choosing the Fastest Subflow Scheduler (CFS)

The schedulers studied in the previous section generally show better performance than the default MPTCP scheduler by predicting the path performance beforehand. However, due to the dynamic Internet traffic and the recent increase of mobile and wireless traffic, the path performance tends to fluctuate over time [7]. In result, the prediction can become inaccurate, thus resulting in performance degradation. In this section, we present our proposed MPTCP scheduler called CFS that overcomes this problem.

3.1 CFS Algorithm

When sending data via MPTCP, data is first stored in the connection-level send buffer, then after being assigned to each subflow, will be assigned to the subflow-level send buffer. Therefore, if the size of the connection-level send buffer is larger than the sum of the send buffers at each subflow-level, there are remaining data packets that are not yet assigned to the send-level buffers.

Algorithm CFS scheduler

```
1.  Find fastest subflow $x_f$ with smallest RTT
2.  if $x_f$ is available for packet transfer then
3.      if SendBuffer is empty then
4.          retransmit packets in the $x_s$ through $x_f$
5.      end if
6.  return $x_s$
7.  else
8.      select $x_s$ using MPTCP default scheduler
9.      return $x_s$
10. end if
```

The above algorithm describes the CFS scheduler in detail. The main principle of the CFS scheduler is similar with the existing minRTT scheduler. However, if a fast subflow is available and there is no more data in the current connection-level send buffer, the CFS scheduler will duplicate the data previously given to the slower subflow, and send it to the fast subflow. This redundant transmission reduces the overall flow completion time, which can improve performance for delay-sensitive applications. Furthermore, this scheme is immune to the link quality fluctuation since it does not require any prediction on each subflow path. In general, if the number of flows is N, the above algorithm gives a computational complexity of O(N).

Proof CFS scheduler

$RTT_f < RTT_s, k$ is a byte from connection-level (not allocated subflow)

$$(1)\ \frac{k}{CWND_f} X\ RTT_f < \frac{k}{CWND_s} X\ RTT_s$$

$$(2)\ \frac{RTT_f}{CWND_f} < \frac{RTT_s}{CWND_s}$$

Note that CFS does not require computation or prediction as in ECF [6]. The main reasons are as follows. First, RTT indicates the time between the sent packet and the received ACK. Therefore, the RTT of a slow subflow must be larger than that of the faster subflow. Next, Eq. (1) compares the time it takes to transmit the send buffer's remaining data of the connection level by the *cwnd* of each subflow. The *cwnd* increases with each received ACK, and, in the case of a slow subflow, the RTT is larger than the RTT of the fast subflow. Thus it is difficult to exceed its *cwnd*. Therefore, when the faster subflow is used, per Eq. 2, the entire flow completion time can be reduced by repetitively transmitting the packet given to the slower subflow.

3.2 Numerical Analysis

We give some numerical analysis to show the effectiveness of our proposal. CFS gives benefit only when the RTT of the fast subflow (RTT_f) is smaller than that of the slow subflow (RTT_s) as shown in the previous subsection. We give the RTTs of the slow and fast subflows as random variables following the *exponential distribution*: X_s and X_f, with mean values of $1/\mu_s$ and $1/\mu_f$, respectively. We compute the probability when the RTT_s may become faster that RTT_f as follows:

$$\Pr(X_s < X_f) = \int_0^\infty \Pr(X_s < y)\, f_f(y) dy \tag{1}$$

where $f_f(y)$ represents the *probability density function* (*pdf*) of X_f. The above equation (1) can be solved as follows:

$$\Pr(X_s < X_f) = \int_0^\infty (1 - e^{-\mu_s y})(\mu_f e^{-\mu_f y})\, dy \ = \frac{\mu_s}{\mu_s + \mu_f} \tag{2}$$

We plot the results in Fig. 6. We set the $1/\mu_f$ =20ms with varying $1/\mu_s$, from 30ms to 300ms. This is consistent with the experimental settings in Section 4. The results show that the probability of the slow subflow overtaking the fast subflow is relatively low, especially in practical conditions where the LTE's RTT is much larger than that of Wi-Fi [3]. When the LTE's RTT is 200ms, then the probability becomes less than 10%. In summary, there is a very high probability that the duplicate transmission of the fast path will complete before the slow path, showing the benefit of CFS.

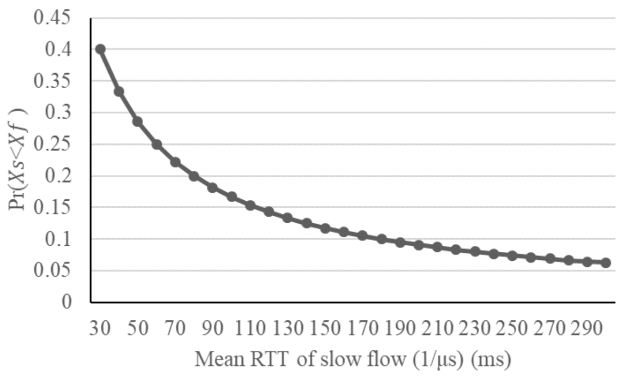


Fig. 6. Probability of $RTT_s < RTT_f$ as a function of $1/\mu_s$.

3.3 Discussions

The potential drawback of our approach is when there is no advantage in duplicating and transmitting the last packets to the fast path. Specifically, when the completion time of the slow path is shorter than the duplicate transmission of the fast path, then the CFS mechanism would have no benefit. The drawback is that the several final packets have been transmitted twice with no edge.

First, according to the analytical results of Section 3.2, there is a very high probability that the duplicate transmission of the fast path will complete before the slow path. So CFS will be advantageous in most cases. Second, even though CFS may transmit duplicate packets in vain, albeit with small probability, this causes minimum damage since it is only done for the final few packets of the flow. This argument is also verified in the experimental results shown in Section 5.

4. Experimental Setup

In this section, we present the experimental setup. We used the Hansung U54X notebook as the MPTCP client and installed Ubuntu Linux 14.04, which built the MPTCP version 0.89 on the Linux kernel, for the evaluations [12]. The server is also built with MPTCP kernel version 0.89 on Ubuntu Linux 14.04 on a general PC. The experimental network topology is shown in in **Fig. 7**.

The MPTCP client is connected to two interfaces, namely Wireless LAN (WLAN) and LTE. WLAN uses the IEEE 802.11ac [15] interface, and the LTE is emulated by using NETEM (i.e., network emulation) [16] over Ethernet. We have also used NETEM to adjust the rate and RTT of both WLAN and LTE as shown in **Table 1**. Similar to our settings, the RTT variation can be very large in real life over time [7]. We have implemented ECF [6] on each client and server [17], as well as our own proposal, CFS for comparison purposes.

The experimental method is as follows. In the first experiment, we use synthetic packet generation; flows of various sizes, 8 types, were randomly generated by the client using socket programming, and flow completion time at the server was measured and compared using each scheduler. We further classify the flow types into 4 types of small flows and 4 types of long flows. In the second experiment, we stored the actual Web page on the MPTCP server and measured and compared the time taken by the client to request it after using Linux's *wget* application [18]. We measured and compared the flow completion time taken of the client to request the web page. The final experiment is similar to that of *wget*. We compared the actual performance with the *curl* application [19] used to parse the web page.

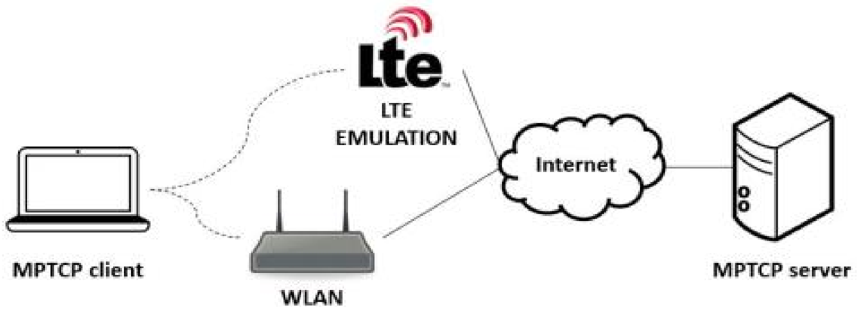


Fig. 7. MPTCP experimental setup

Table 1. Rate and RTT controlled by NETEM

	Rate	RTT	RTT Variation
ETHERNET (LTE Emulation)	150 mbit	200 ms	±150 ms
WLAN	500 mbit	20 ms	±15 ms

5. Performance Evaluation

5.1 Synthetic Packet Generation Experiment

Clients synthetically generated flows of different sizes and sent them to the server. Then, it measured the overall flow response time. **Fig. 8** shows the short-flow of flow sizes of 64 KB, 128 KB, 256 KB, 512 KB, and 1 MB, repeating 30 times for each flow size.

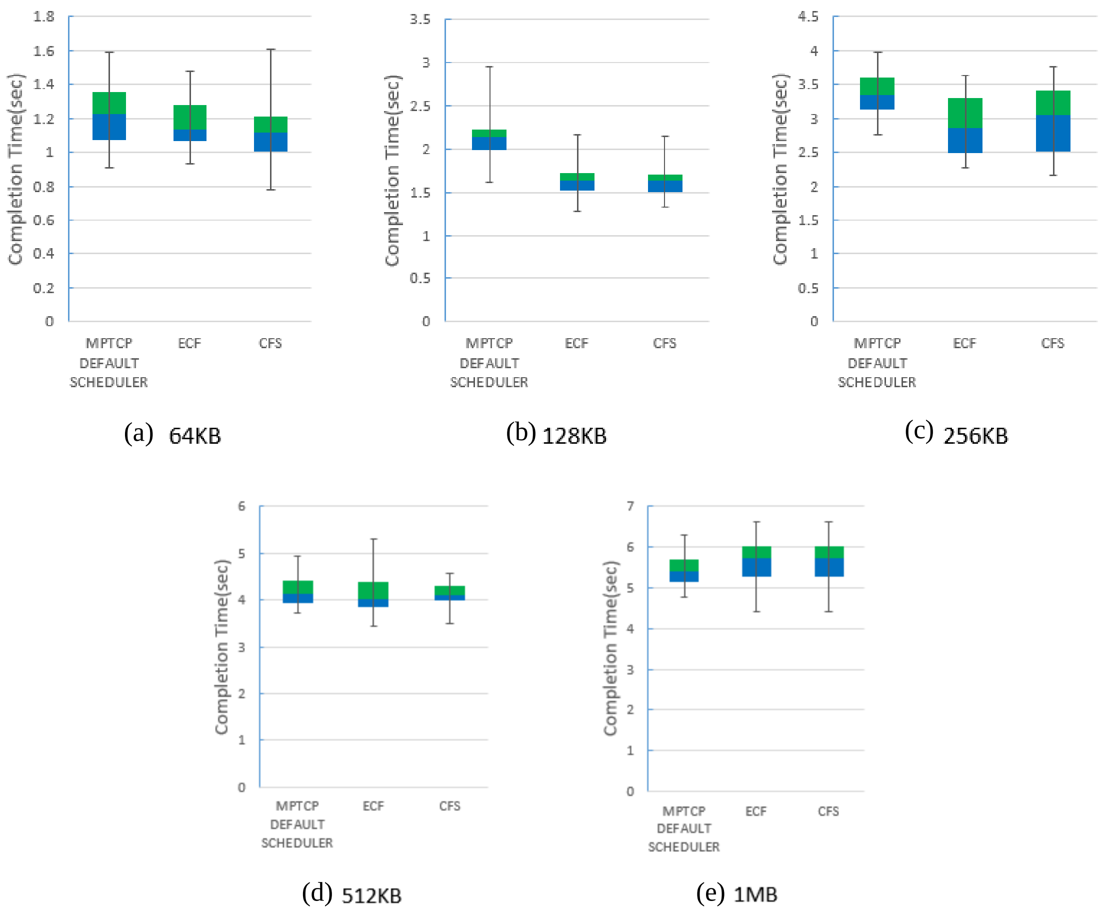


Fig. 8. Completion time for short flows - various traffic size (s)

In **Fig. 8**, the horizontal axis represents each MPTCP scheduler and the vertical axis represents the flow completion time. The lower the flow completion time, the faster the response time for the application user. Overall, CFS has a low completion time compared to the default MPTCP scheduler [12] and ECF [6]. If the latency difference between the two paths is large and the network performance changes frequently over time, the predicted value may be inaccurate for ECF. By utilizing the characteristic of these paths, the CFS lowers the completion time by duplicating the slow subflows' data onto the fast subflows.

Table 2. Average completion time for short flows (sec)

	64 KB	**128 KB**	**256 KB**	**512 KB**	**1 MB**
DEFAULT	1.23	2.15	3.33	4.17	5.44
ECF	1.16	1.66	2.92	4.14	5.63
CFS	1.11	1.64	2.96	4.11	5.44

Table 3. Reduced percentage of average completion time over the default scheduler for short flows

	64 KB	**128 KB**	**256 KB**	**512 KB**	**1 MB**
ECF	5%	23%	12%	1%	-4%
CFS	10%	24%	11%	1%	0%

Table 2 shows the average value of each experiment shown in **Fig. 8**. Furthermore, **Table 3** shows the reduced percentage of completion time for ECF and CFS over the minRTT scheduler. As shown in **Table 3**, the performance is similar or slightly higher than that of ECF. In particular, when the flow size is 256KB.

We note that the main focus of our work is to reduce the completion time of the flows. Thus, the completion time is meaningful for relatively short flows. However, we next conduct experiments on the large flows (i.e., 4 MB, 8 MB, 16 MB, 32 MB), similar to the work in [3]. In the case of large flows, as the traffic increases, it may cause redundant transmission reducing the overall performance. However, at the end of the flow, there is no impact on performance, because it is redundant only if there is no more data to send to the send buffer at the connection level. Our argument is shown in **Fig. 9**. There is not much difference between performance of the existing scheduler and the ECF. The average values for these experiments are shown in **Table 4**.

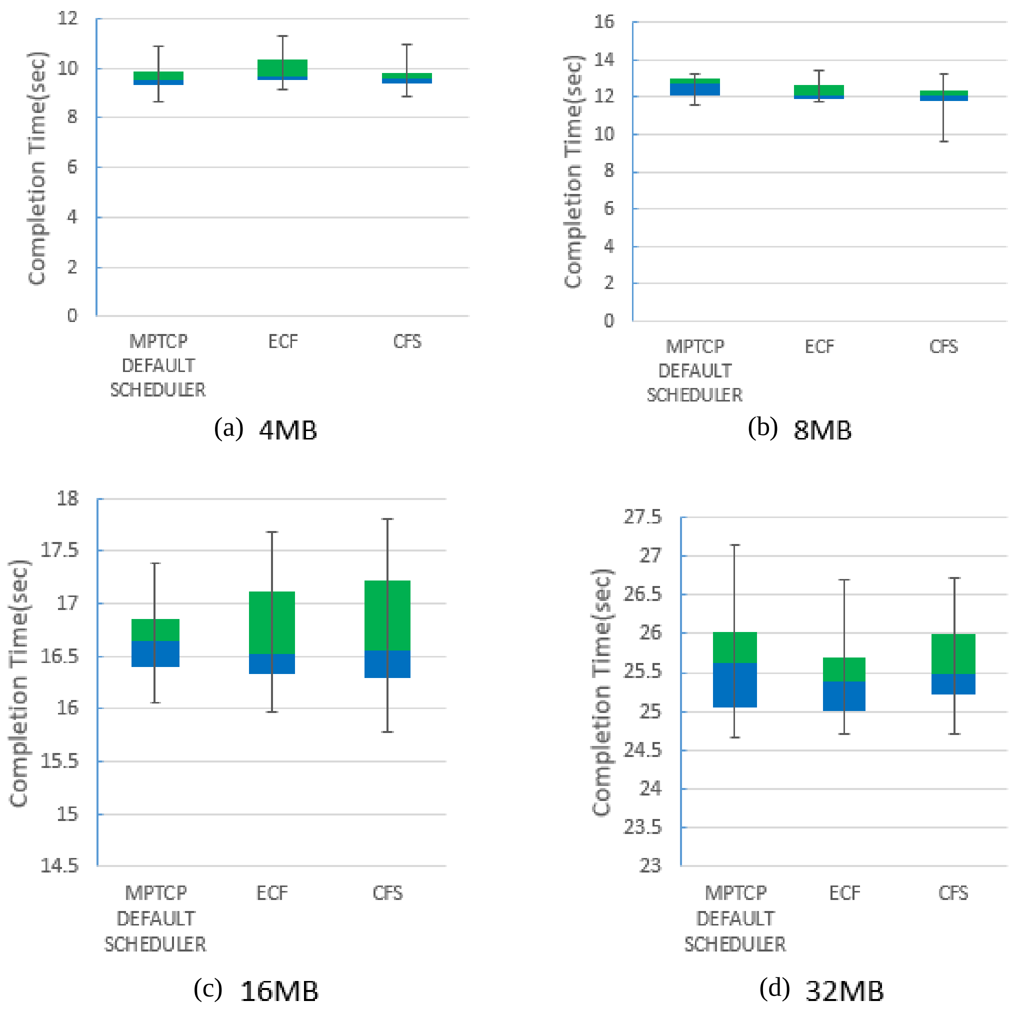


Fig. 9. Completion time for large flows- various traffic size

Table 4. Average completion time for large flows (sec)

	4 MB	8 MB	16 MB	32 MB
DEFAULT	9.67	12.54	16.66	25.62
ECF	9.89	12.30	16.67	25.44
CFS	9.68	12.10	16.72	25.55

5.2 Actual Web Page Request Experiment

In this experiment, we use real web applications for performance comparisons. We measured the total flow completion time when the client requests the Web pages from the MPTCP enabled web server. The MPTCP server's actual web page data set is shown in **Table 5**.

Table 5. Web file size

Filename	**File Size**
Wikipedia	76 KB
Google	220 KB
Amazon	*742 KB*
Huffington Post	1,146 KB

We then measured the overall flow completion time using two applications (i.e., *wget*, *curl*), used when retrieving web pages or data from a Linux server.

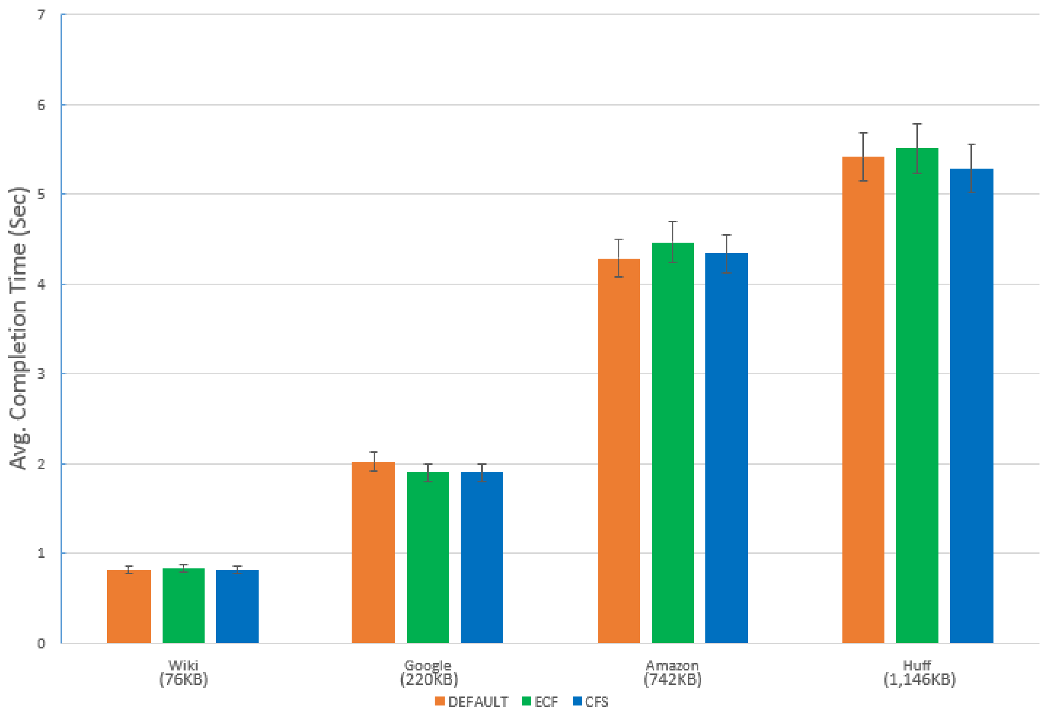


Fig. 10. Objection download average completion time - various file size (sec)

Table 6. Object download average completion time (sec)

	Wikipedia	**Google**	**Amazon**	**Huffington Post**
DEFAULT	0.83	2.02	4.29	5.42
ECF	0.84	1.91	4.47	5.52
CFS	0.81	1.91	4.34	5.29

First, we measured flow completion time of each web page stored on the web server using *wget* [18]. *Wget* is a free software package for retrieving files using widely used Internet protocols such as HTTP and HTTPS. It shows a traffic pattern similar with the general web-surfing via a web browser. The experiment was repeated 30 times for each web page. **Fig. 10** shows the flow completion time (i.e., time between the web page request and receipt of data) using each MPTCP scheduler. The horizontal axis represents each MPTCP scheduler, and the vertical axis represents the average value of the flow completion time 30 tests. Overall, the CFS scheduler has a lower flow completion time than the minRTT scheduler and ECF. **Table 6** summarizes the mean values for the figures. The table also shows that CFS has slightly better performance than the minRTT or ECF.

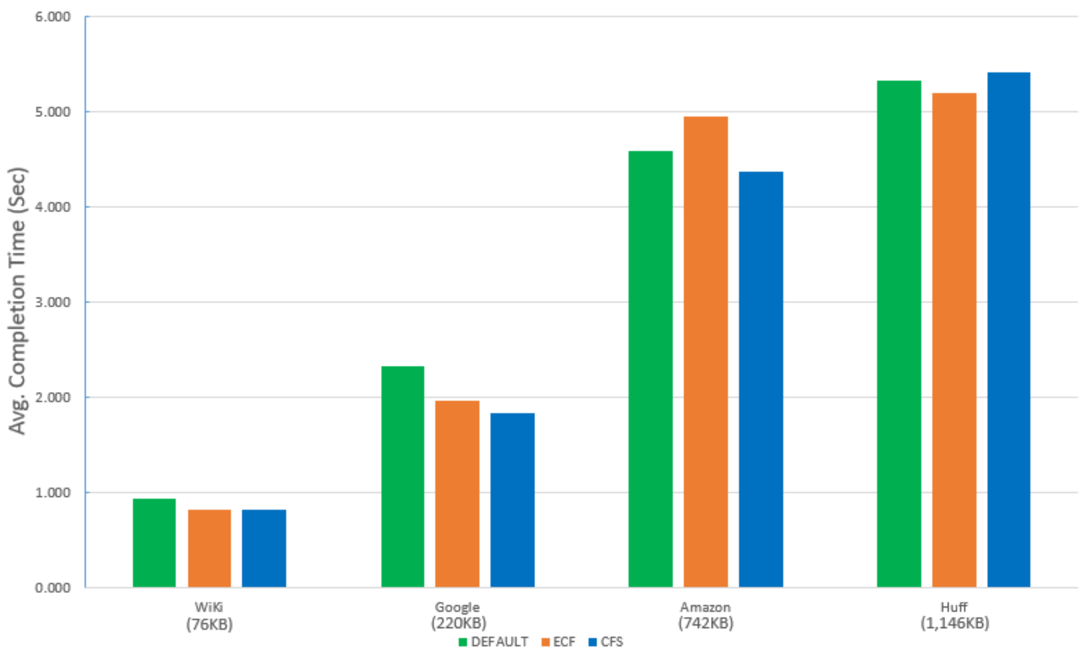


Fig. 11. Web-parsing average completion time - various file size

Table 7. Web parsing average completion time (sec)

	Wikipedia	**Google**	**Amazon**	**Huffington Post**
DEFAULT	0.93	2.33	4.58	5.32
ECF	0.82	1.97	4.95	5.19
CFS	0.81	1.83	4.36	5.41

Next, we measured the flow completion time using *curl* [19]. *Curl* is similar to *wget*, that it downloads data, but *curl* supports libraries and various protocols. The experiment was repeated 30 times for each web page. **Fig. 11** shows the overall flow completion time of parsing the web page. The horizontal axis represents each scheduler and the vertical axis represents the average flow completion time of each scheduler. The minRTT scheduler and ECF have a slightly higher flow completion time than CFS. **Table 7** summarizes their mean values.

In summary, CFS outperforms the previous schedulers for real web traffic as well as synthetic traffic, due to its ability to improve the completion time without the delay prediction.

6. Conclusion

In this paper, we proposed a new MPTCP scheduler, called choose fastest subflow (CFS) scheduler, that copes well in general environments where the two paths have diverse and dynamic performances. CFS utilizes the characteristics of these paths to reduce the overall flow completion time by redundantly sending the last part of the flow to both paths. We present some numerical analysis to show that, there is a very high probability that the duplicate transmission of the fast path will complete before the slow path, showing the benefit of CFS.

We conducted evaluation through two types of real experiments, synthetic packet generation and actual Web page requests. We compared CFS with the current MPTCP implementation on the Linux kernel, and also with the state-of-art scheduler, ECF. The experimental results showed that CFS consistently outperforms the previous proposals in all cases.

References

[1] Use Multipath TCP to create backup connections for IOS. https://support.apple.com/en-gb/HT201373

[2] KT's MPTCP Proxy Experiences. https://www.ietf.org/proceedings/91/slides/slides-91-mptcp-5.pdf

[3] Y.-C. Chen, Y. Lim, R. J. Gibbens, E. M. Nahum, R. Khalili, and D. Towsley, "A measurement-based study of MultiPath TCP performance over wireless networks," in *Proc. of the 2013 conference on Internet measurement conference - IMC '13*, pp. 455-468, 2013. Article (CrossRef Link)

[4] N. Kuhn, E. Lochin, A. Mifdaoui, G. Sarwar, O. Mehani, and R. Boreli, "DAPS: Intelligent delay-aware packet scheduling for multipath transport," in *Proc. of IEEE International Conference on Communications (ICC)*, pp. 1222-1227, 2014. Article (CrossRef Link)

[5] S. Ferlin, O. Alay, O. Mehani, and R. Boreli, "BLEST: Blocking estimation-based MPTCP scheduler for heterogeneous networks," in *Proc. of IFIP Networking Conference (IFIP Networking) and Workshops*, pp. 431-439, 2016. Article (CrossRef Link)

[6] Y. Lim, E. M. Nahum, D. Towsley, and R. J. Gibbens, "ECF: An MPTCP Path Scheduler to Manage Heterogeneous Paths," in *Proc. of the 13th International Conference on emerging Networking EXperiments and Technologies - CoNEXT '17*, pp. 147-159, 2017. Article (CrossRef Link)

[7] T. H.-Jorgensen, B. Ahlgren, P. Hurtig, A. Brunstrom, "Measuring Latency Variation in the Internet," in *Proc. of the 12th International on Conference on emerging Networking EXperiments and Technologies*, pp. 473-480, 2016. Article (CrossRef Link)

[8] S. Deng, R. Netravali, A. Sivaraman, and H. Balakrishnan, "Wi-Fi, LTE, or Both: Measuring Multi-Homed Wireless Internet Performance?," in *Proc. of the 2014 Conference on Internet Measurement Conference - IMC '14*, pp. 181-194, 2014. Article (CrossRef Link)

[9] S. Barré, C. Paasch, and O. Bonaventure, "MultiPath TCP: From Theory to Practice," *Lecture Notes in Computer Science*, pp. 444–457, 2011. Article (CrossRef Link)

[10] HanArr Ko, JaeWook Lee, SangHeon Baek, JaeHyun Hwang. "Multipath TCP (MPTCP) Standardization and Technology Development Trend," *The Journal of The Korean Institute of Communication Sciences*, pp. 9-16, 2014. Article (CrossRef Link)

[11] C. Paasch, S. Ferlin, O. Alay, and O. Bonaventure, "Experimental evaluation of multipath TCP schedulers," in *Proc. of the 2014 ACM SIGCOMM Workshop on Capacity-Sharing Workshop - CSWS '14*, pp. 27-32, 2014. Article (CrossRef Link)

[12] Multipath TCP-Linux Kernel Implementation. https://www.multipath-tcp.org

[13] C. Raiciu, C. Paasch, S. Barre, A. Ford, M. Honda, F. Duchene, O. Bonaventure, M. Handley, "How hard can it be? Designing and implementing a deployable multipath TCP," in *Proc. of Networked System Design and Implementation – NSDI'12*, 2012. Article (CrossRef Link)

[14] J. Hwang, and J. Yoo, "Packet Scheduling for Multipath TCP," in *Proc. of International Conference on Ubiquitous and Future Networks - ICUFN'15*, 2015.

[15] IEEE 802.11ac. Standard for Information Technology - Telecommunications and Information Exchange Between Systems - Local and Metropolitan Area Networks - Specific Requirements - Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications Amendment 4: Enhancements for Very High Throughput for Operation in Bands Below 6 GHz.

[16] Network Emulation NETEM. https://wiki.linuxfoundation.org/networking/netem

[17] Multipath TCP ECF Scheduler Source Code. https://people.cs.umass.edu/~ylim/mptcp_ecf

[18] GNU Wget. https://www.gnu.org/software/wget

[19] Curl. https://curl.haxx.se

[20] W. Deng, R. Yao, H. Zhao, X. Yang, G. Li, "A novel intelligent diagnosis method using optimal LS-SVM with improved PSO algorithm," *Soft Computing*, 23.7. pp. 2445-2462, 2017. Article (CrossRef Link)

[21] W. Deng, H. Zhao, X. Yang, J. Xiong, M. Sun, B. Li, "Study on an improved adaptive PSO algorithm for solving multi-objective gate assignment," *Applied Soft Computing*, 59, pp. 288-302, 2017. Article (CrossRef Link)

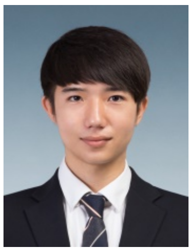
[22] W. Deng, H. Zhao, L. Zou, G. Li, X. Yang, D. Wu, "A novel collaborative optimization algorithm in solving complex optimization problems," *Soft Computing*, 21(15), pp. 4387-4398, 2017. Article (CrossRef Link)

[23] W. Deng, S. Zhang, H. Zhao, X. Yang, "A novel fault diagnosis method based on integrating empirical wavelet transform and fuzzy entropy for motor bearing," *IEEE Access*, 6(1), pp. 35042-35056, 2018. Article (CrossRef Link)

[24] H. Zhao, R. Yao, L. Xu, Y. Yuan, G. Li, W. Deng, "Study on a novel fault damage degree identification method using high-order differential mathematical morphology gradient spectrum entropy," *Entropy*, 20(9), pp. 682, 2018. Article (CrossRef Link)

[25] H. Zhao, M. Sun, W. Deng, X. Yang, "A new feature extraction method based on EEMD and multi-scale fuzzy entropy for motor bearing," *Entropy*, 19(1), pp. 14, 2017. Article (CrossRef Link)

[26] Y Liu, X Yi, R Chen, Z Zhai, J Gu, "Feature extraction based on information gain and sequential pattern for English question classification," *IET Software*, 12(6), pp. 520-526, 2018.
Article (CrossRef Link)



Geonyeoung Heo (M.S) received his B.S. in Computer Engineering, and M.S. in Software from Gachon University in 2016 and 2018, respectively. In 2018, he has joined m2soft as a researcher. His research interests include Multipath TCP, and wireless networks.

Joon Yoo received his B.S. in Mechanical Engineering from KAIST, and PhD in Computer Science and Engineering from Seoul National University in 1997 and 2009, respectively. From 2009 to 2010, he worked as a postdoctoral researcher at the University of California, Los Angeles, and from 2010 to 2012, he worked at Bell Labs Seoul, Korea as a Member of Technical Staff. He is currently with the Department of Software, Gachon University, Korea, as an associate professor since 2012. His research interests include vehicular networks, data center networks, and IEEE 802.11 WLAN.