Why Multipath TCP Degrades Throughput Under Insufficient Send Socket Buffer and Differently Delayed Paths

Toshihiko Kato, Adhikari Diwakar, Ryo Yamamoto, and Satoshi Ohzahata
Graduate School of Informatics and Engineering
University of Electro-Communications
Tokyo, Japan
e-mail: kato@net.lab.uec.ac.jp, diwakaradh@net.lab.uec.ac.jp, ryo-yamamoto@uec.ac.jp, ohzahata@uec.ac.jp

Abstract— **Recently, the Multipath Transmission Control Protocol (MPTCP) comes to be used widely. It allows more than one TCP connections via different paths to compose one Multipath TCP communication. Our previous papers pointed out that insufficient send socket buffer makes the throughput worse than that of single path TCP, when the subflows have different transmission delays. Although our previous papers gave the detailed analysis on the throughput degradation focusing on the relationship between the send socket buffer size and the delay, they did not clarify the reason of the throughput degradation. This paper investigates the Linux MPTCP software and the MPTCP communication details, and clarifies why the insufficient socket buffer degrades the MPTCP throughput.**

Keywords- multipath TCP; send socket buffer; head-of-line blocking.

I. INTRODUCTION

Recently, mobile terminals with multiple interfaces have come to be widely used. For example, most smart phones are installed with interfaces for 4G Long Term Evolution (LTE) and Wireless LAN (WLAN). In order for applications to use multiple interfaces effectively, Multipath TCP (MPTCP) is being introduced in several operating systems, such as Linux, Apple OS/iOS and Android. MPTCP is defined in three RFC documents by Internet Engineering Task Force. RFC 6182 [1] outlines the architecture guidelines for developing MPTCP protocols. It defines the ideas of *MPTCP connection* and *suflows* (TCP connections associated with an MPTCP connection). RFC 6824 [2] presents the details of extensions to the traditional TCP to support multipath operation. It defines the MPTCP control information realized as new TCP options, and the MPTCP protocol procedures. RFC 6356 [3] presents a congestion control algorithm that couples those running on different subflows.

MPTCP has some problems when subflows are established over heterogeneous paths with different delay, such as an LTE network and a WLAN. TCP ACKnowledgment (ACK) segments from a path with longer delay return later than those from a shorter delay path. This causes a *Head-of-Line* (*HoL*) *blocking*, in which TCP data segments over a longer delay subflow block the window sliding while waiting for their ACKs [4]. In order to avoid this problem, the selection of the appropriate subflow is required. The function to select a subflow for transferring a data segment is called a *scheduler*, and several scheduler algorithms have been proposed so far. Originally, MPTCP implementation adopted the lowest Round-Trip Time (RTT) first and the round-robin schedulers [5]. However, both of them suffer from the HoL blocking. The opportunistic Retransmission and Penalization mechanism (*RP mechanism*) [6] [7] is used in the current MPTCP implementation as a default. When a data sender detects that new data cannot be sent out due to an HoL blocking over a specific subflow, it retransmits the oldest unacknowledged data through a subflow with the lowest RTT (opportunistic retransmission). At the same time, the subflow that occurred this HoL blocking is punished by halving its congestion window (penalization).

The Delay Aware Packet Scheduling (DAPS) [8] and the Out-of-order Transmission for In-order Arrival Scheduling (OTIAS) [9] take account of subflow delays and schedule data segment sending for in-order receiving. The BLocking ESTimation scheduler (BLEST) [10] estimates whether a subflow will cause an HoL blocking and dynamically adapts scheduling to prevent blocking.

Those schedulers improve the MPTCP performance compared with the original scheduler algorithm, and several studies report the results of MPTCP performance evaluation through heterogeneous paths [11]-[15]. However, those proposals of schedulers and the performance evaluation reports are focusing only on the receive socket buffer. While insufficient receive socket buffer invokes HoL blocking, the send socket buffer also gives some impacts on the TCP throughput.

In our previous papers [16] [17], we pointed out that an insufficient size of send socket buffer provokes more serious throughput degradation than insufficient receive socket buffer. Although our previous papers analyzed the detailed behaviors of MPTCP by investigating the MPTCP and TCP level sequence numbers and windows, they did not discuss why such performance degradation happens. In this paper, we clarify the reason by investigating the Linux MPTCP software and the communication traces.

The rest of paper consists of the following sections. Section 2 shows the details of MPTCP data transfer procedure and the related work on the MPTCP scheduler. Section 3 gives the results of performance evaluation in the case that an MPTCP connection provides poor throughput than a single TCP connection due to the insufficient send socket buffer. Section 4 shows the behavior of Linux MPTCP software in the case of limited send buffer, and discusses the reason of the performance degradation. Section 5 concludes this paper.

II. RELATED WORK

This section describes the related work of our work.

A. MPTCP Data Transfer Procedures

As described in Figure 1, the MPTCP module is located on top of TCP. MPTCP is designed so that the conventional applications do not need to care about the existence of MPTCP. MPTCP establishes an MPTCP connection that is associated with two or more regular TCP connections called subflows. The management and data transfer over an MPTCP connection is done by the TCP options newly introduced for MPTCP operation. In the beginning, one subflow and an MPTCP connection are established through the TCP three way handshake using a Multipath Capable (MP_CAPABLE) TCP option. Next, another subflow is established and associated with the existing MPTCP connection by use of a Join Connection (MP_JOIN) option. This option contains the identification of the MPTCP connection to be joined.

After establishing multiple subflows, MPTCP takes one input data stream from a sender-side application, splits it into subflows, and reassembles the split data streams at the receiver side. The MPTCP connection level maintains the data sequence number independent of the subflow level sequence numbers. The data and ACK segments used in a subflow may contain a Data Sequence Signal (DSS) option depicted in Figure 2. The number is assigned on a byte-by-byte basis similarly with the TCP sequence number. The value of data sequence number is the number assigned to the first byte conveyed in that TCP segment. The data sequence number, subflow sequence number (relative value) and data-level length define the mapping between the MPTCP connection level and the subflow level. The data ACK is analogous to the behavior of the standard TCP cumulative ACK, and specifies the next data sequence number a receiver expects.

We need to say that there is no window size field in the DSS option. Instead, the window size contained in the TCP header is used for the flow control in MPTCP. That is, the flow control is performed for the data sequence number at the MPTCP connection level as well as the subflow level. The control for the data sequence number is done in a way that the data sequence number is not more than the data ACK plus the window size. It is also required that the upper window edge in the MPTCP connection level (the data ACK plus the window size) does not shrink. In order to fully utilize the capacity of all subflows, a receiver needs to provide the following buffer space so that a sender can keep all subflows fully utilized [2].

$$\text{Buffer size} = \sum_{i=1}^{n} bw_i \times RTT_{max} \times 2. \qquad (1)$$

Here, n is the number of subflows, bw_i is the bandwidth of subflow i, and RTT_{max} is the highest RTT among all subflows. This equation means that an MPTCP connection can send data segments at full speed during the highest RTT, even if a loss event occurs.

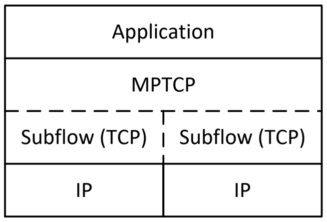


Figure 1. MPTCP layer structure.

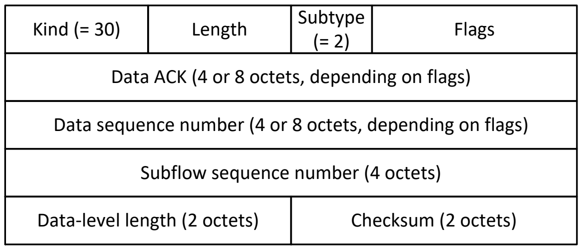

Kind (= 30)
Length
Subtype (= 2)
Flags
Data ACK (4 or 8 octets, depending on flags)
Data sequence number (4 or 8 octets, depending on flags)
Subflow sequence number (4 octets)
Data-level length (2 octets)
Checksum (2 octets)

Figure 2. Data sequence signal option.

B. Related Work on MPTCP Scheduler

As stated in the previous section, the mechanism to assign data to multiple subflows is called a scheduler. The MPTCP implementation for Linux operating system supports the default scheduler, the lowest RTT first with RP mechanism, which is called *minRTT*. The minRTT scheduler first sends data over one subflow with the lowest RTT until its window is full. Then, it starts transmitting over the next subflow. When it detects an HoL blocking, it retransmits the oldest data as a new data segment over the lowest RTT subflow, and halves the congestion window of the subflow that caused this blocking.

The other schedulers mentioned in the previous section can be summarized as follows. DAPS aims for in-order arrival at the receiver to prevent its buffer from blocking [8]. It sends data segments in inverse proportion to the delay of individual subflows with the strategy that the younger numbered data segments are transferred through the path with shorter delay. OTIAS provides the out-of-order transmission at the sender for the in-order arrival at the receiver [9]. One of the difference between DAPS and OTIAS is that DAPS focuses on the scheduling of multiple packets, but OTIAS tries to determine the scheduling of only one packet. BLEST, on the other hand, takes a proactive stand towards minimizing HoL blocking [10]. Rather than penalizing the slow subflows, it estimates whether a path will cause HoL blocking and dymamically adapts scheduling to prevent it.

Several publications discussed the performance evaluation [6] [7] [11][-15]. The early stage papers [6] [7] [11] [12] focus mainly on measuring MPTCP throughput with changing the receive socket buffer size. Kim et al. [13] proposes a scheduler using the buffer blocking prediction based on receive buffer size and RTT. It gives the time variations of throughput as a performance evaluation. Zhou et al. [14] shows the MPTCP performance evaluation over the real Internet by changing socket buffer size under the assumption that the sizes of send and receive socket buffers are the same. Dong et al. [15] compares the performance of several schedules including DAPS, OTIAS, and BLEST. In the performance evaluation, it measured the file transfer completion time by changing RTT ratio, receive socket buffer size, and transferred file size.

Those papers have two problems. First, all of them use only macroscopic performance metrics, such as the average throughput and the completion time for a file transfer. They do not discuss the detailed performance analysis taking account of the MPTCP parameters, such as data sequence number and data ack number. Secondly, they mainly focus on the receive socket buffer. However, the send socket buffer also gives some impacts on the TCP throughput.

In contrast, we took a microscopic approach that analyzes the detailed MPTCP internal behaviors in the performance evaluation, and focused on send socket buffer size as well as receive buffer size. In [16], we evaluated the MPTCP performance by changing receive and send socket buffer sizes independently, and clarified that an insufficient send socket buffer provokes more serious throughput degradation than insufficient receive socket buffer. Kato et al. [17] provided more detailed analysis of the MPTCP communicatison under insufficient send socket buffer through dissimilarly delayed subflows.

III. Throughput Degradation due to Insufficient Send Socket Buffer

As for the case that send socket buffer is insufficient, our previous papers showed the following results when MPTCP communication is done by two subflows whose transmission delay is different; fast subflow and slow subflow. In the case that the send socket buffer size is small, the communication is done through only the fast subflow. As the send socket buffer size increases, the communication is done through two subflows, but the MPTCP connection level throughput is lower than the single path TCP communication whose delay is equal to the fast subflow. When the send socket buffer size becomes bigger, two subflows are used in the MPTCP communication and its throughput is larger than the bandwidth of one subflow.

In this section, we show the detailed MPTCP behaviors when the MPTCP connection level throughput is lower than a single path TCP throughput.

A. Experimental Settings

In the experiment, we use the network configuration of shown in Figure 3. Two hosts running the Linux operating system (Ubunts 16.04 LTS), data sender and receiver, are connected together through two 1Gbps Ethernet links. The private IP addresses are assigned as shown in the figure. In one Ethernet link, a network emulator is inserted in order to provide delay. Although the physical data rate is 1Gbps, the frame transmission speed is limited to 100 Mbps using Linux traffic control (*tc*) command. We establish one MPTCP connection between two hosts with two subflows, one between 192.168.0.1 and 192.168.0.2 (called *fast subflow*) and the other between 192.168.1.1 and 192.168.1.2 (called *slow subflow*). The scheduler is the default one, minRTT. By consulting the results in our previous papers, the delay inserted at the network emulator is set to 40 msec for one direction, i.e., 80 msec in round-trip. The send socket buffer size is set to 1,048,576 bytes (1 Gibibytes) for the minimum, default, and maximum sizes, using the `sysctl -w net.ipv4.tcp_wmem` command. The receive socket buffer at the receiver uses the default value, which is 4,096, 87380, and 6,291,456 bytes for the minimum, default, and maximum sizes, respectively.

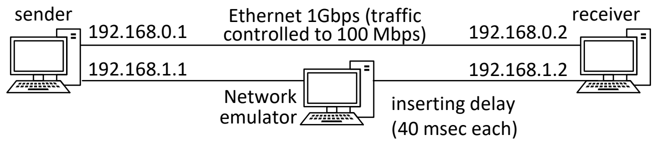

sender
192.168.0.1
Ethernet 1Gbps (traffic controlled to 100 Mbps)
192.168.0.2
receiver
192.168.1.1
192.168.1.2
Network emulator
inserting delay (40 msec each)

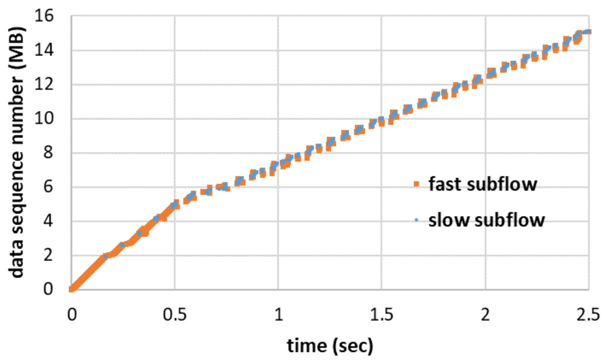
Figure 3. Network configuration for evaluation.

In the performance evaluation, *iperf* is used in both hosts and bulk data transfer is executed for 10 seconds. During the bulk data transfer, packet traces are collected at the sender side by use of *tcpdump*. Those traces are examined in detail with *Wireshark*, a network protocol analyzer whose version is 3.2.4. We also execute *tcpprobe* in the sender side in order to collect TCP related information like congestion window size during data transfer.

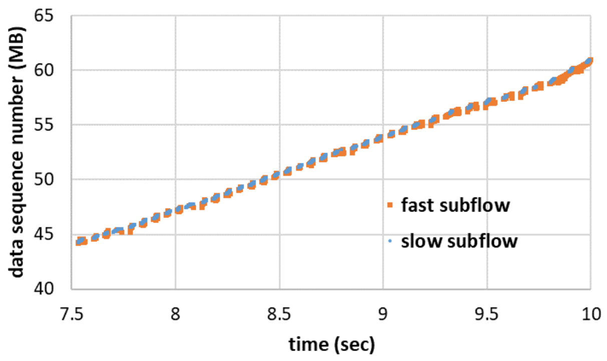
B. Results and Analysis

We executed five experiment runs under the conditions described above. The throughput measured at the receiver side ranges from 42.4 Mbps to 49.8 Mbps. The average is 46.8 Mbps and the standard deviation is 3.54 Mbps. Since the link bandwidth of the fast subflow is 100 Mbps, the MPTCP connection level throughput is lower than the subflow bandwidth.

Hereafter, we pick up the forth experiment run whose throughput is 48.6 Mbps. Figure 4 shows the time variation



(a) from 0 sec to 2.5 sec.

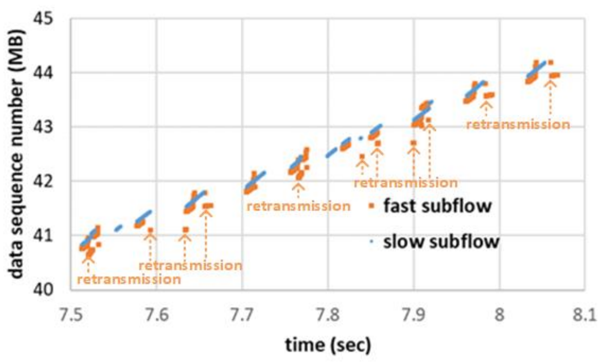


(b) from 7.5 sec to 10 sec.

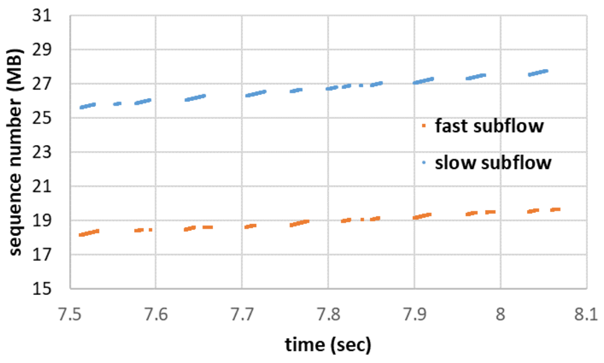
Figure 4. Time variation of data sequence number.

of the data sequence number sent over the fast and slow subflows. In this figure, we show the graph focusing on the results during the first and the last 2.5 secs. The two graphs show similar tendencies. The data sequence number is increasing in both fast and slow subflows. The increase itself is intermittent, that is, the data transmission in this case repeats sending and stopping. This behavior is considered to be the reason for low throughput.

Figure 5 shows the time variation of the MPTCP data sequence number and the TCP sequence number focused on the period between 7.5 sec and 8.1 sec. By zooming in the change of the data sequence number, we can find that there are several retransmissions in the fast subflow, e.g., at 7.55 sec, 7.74 sec, 7.78 sec, some of which are explicitly indicated in the figure. Figure 5(b) shows the increase of TCP sequence numbers in the fast and slow subflows, and there are no retransmissions in both of them. This result means that the retransmissions shown in Figure 5(a) are the opportunistic retransmissions installed in the minRTT scheduler.



(a) data sequence number



(b) TCP sequence number

Figure 5. Detailed behavior between 7.5 sec and 8.1sec.

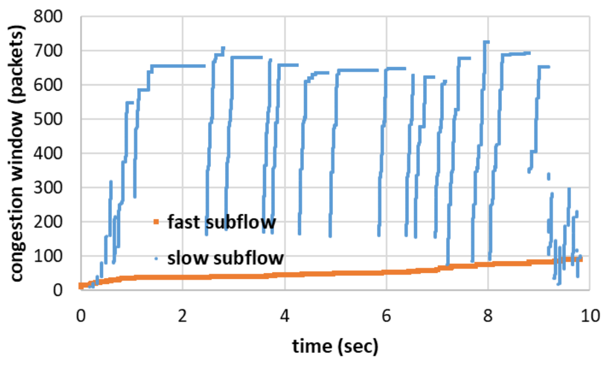


Figure 6. Time variation of congestion window size.

We also measured the values of the congestion window size of the fast and slow subflows, at the timing of sender receiving ACK segments by use of the tcpprobe module. Figure 6 shows the time variation of congestion window size in the fast and slow subflows. The congestion window size of the slow subflow increases up to around 700 packets and undergoes drops more than 20 times. We confirmed that there were no packet losses in the TCP level, and so these drops are caused by the penalization implemented in the minRTT scheduler. The congestion window size in the fast subflow, on the other hand, experienced no drops. The reason that the window is kept in a relatively low value is that the congestion window validation [18] was effective due to small RTT in the fast subflow.

IV. Analysis of Linux MPTCP Software

This section shows the detail of Linux MPTCP software and discusses why the performance degradation happens.

A. *Internals of Linux MPTCP Software*

When an application sends a data using MPTCP, function `tcp_sendmsg_locked()` processes this request. Figure 7 shows its outline. The main part of this function is a while loop for handling data given by an application. In the loop, the buffer status is checked at first. If there is no send socket buffer space, checked by `sk_stream_memory_free()`, or if the socket buffer for a data segment cannot be allocated, done by `sk_stream_alloc_sk()`, then the control is jumped to `wait_for_snnbuf`. Here, this module will wait for some period by `sk_stream_wait_memory()`. On the other hand, if the send socket buffer has enough space, the data is copied to the allocated buffer (`skb_add_data_nocache()`) and transmitted (`__tcp_push_pending_frames()` or

```
tcp_sendmsg_locked(struct sock *sk, msg, size)
    // msg: data to send, size: data size
{
  while(msg_left(msg)) { //loop while msg is left
    if(new_segment) {
      if(!sk_stream_memory_free(sk)) //if no sock buf
        goto wait_for_sndbuf;         //jump
      skb = sk_stream_allock_skb(sk) //allocate buffer
      if(!skb)                       //if allocation failed
        goto wait_for_sndbuf; //jump
    }
    skb_add_data_nocache(sk, skb, msg, size);
        //copy data to socket buffer
    if(forced_push(sk)) {
      __tcp_push_pending_frames(sk);
                          //call mptcp_write_xmit()
    } else if(skb == tcp_send_head(sk))
      tcp_push_one(sk);      //call mptcp_write_xmit()
    continue;  //go to while for next segment
wait_for_sndbuf:
    sk_stream_wait_memory(sk, timer);
  }
  return sent_data_size;
}
```

Figure 7. Outline of tcp_sendmsg_locked().

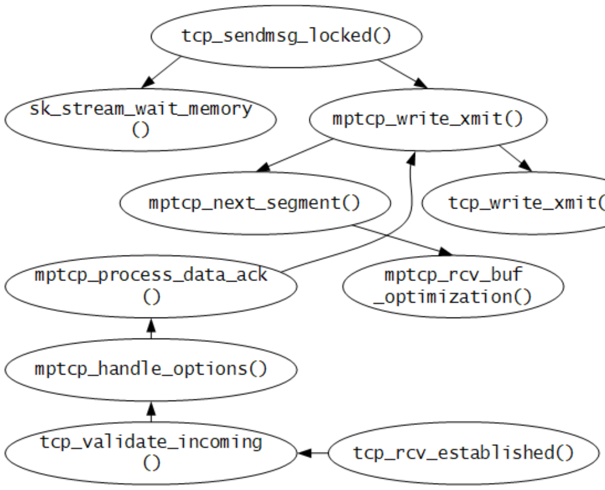


Figure 8. Function calls of data sending and receiving.

`tcp_push_one()`). The latter two functions will eventually call `mptcp_write_xmit()` to perform actual sending out. This program structure tells that the waiting on the send socket buffer shortage follows a different processing path from the sending data segment.

Figure 8 shows the relationship of function calls in the data sending and receiving. As described above, the data sending is handled in `mptcp_write_xmit()`, which calls `mptcp_next_segment()` for obtaining the data segment being sent next, and `tcp_write_xmit()`, which sends the data segment out to the IP module. Within the `mptcp_next_segment()` function, `mptcp_rcv_buf_optimization()` is called to check whether to perform the RP mechanism. When the RP mechanism is performed actually this function returns the buffer addresses that contains the data segment to be transmitted.

This figure also describes the function calls when receiving a data or ACK segment. When the TCP module receives a segment in the ESTABLISHED state, `tcp_rcv_eshtablished()` is called. In this function, the segment itself and its parameters are checked in `tcp_validate_incoming()`, which calls `mptcp_handle_options()` for processing a DSS option. In this function, `mptcp_process_data_ack()` is called for a data ACK parameter, and then this function calls `mptcp_write_xmit()` eventually. In this function, the RP mechanism is performed when it is necessary.

The following two points need to be mentioned. First, the waiting for buffer release in `sk_stream_wait_memory()` and the other processing are completely independent. Especially, the behavior of sending data requested from the upper layer stops completely during this waiting, because `tcp_sendmsg_locked()` is the only function to handle the data send request from the upper layer and it is blocked in this waiting function. The other is that the RP mechanism is applied to the subflows other than the one invoked the `mptcp_rcv_buf_optimization()` function.

B. Behaviors of Linux MPTCP Software

Figure 9 shows the time variation of congestion window sizes in the fast and slow subflows from 7.5 sec to 8.1 sec.

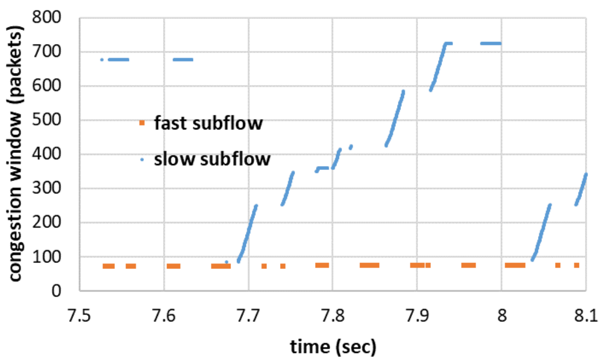


Figure 9. Time variation of congestion window size between 7.5 sec and 8.1 sec.

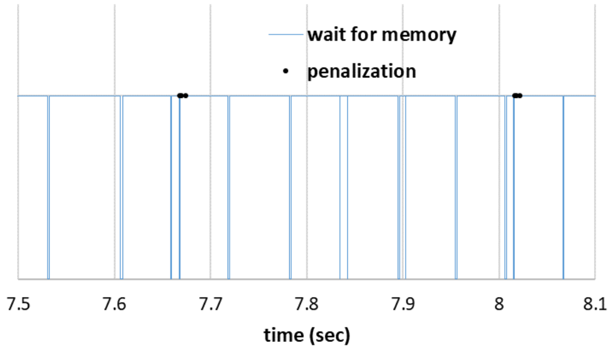


Figure 10. Behavior of wait for memory and penalization.

During this period, the window of the slow subflow is reduced twice, between 7.6 sec and 7.7 sec, and between 8.0 sec and 8.1 sec.

In order to analyze the MPTCP software behavior in Linux, we modified the Linux kernel to generate the timestamps of the entry and exit of `sk_stream_wait_memory()`, and those of the penalization in `mptcp_rcv_buf_optimization()`. Figure 10 shows those results between 7.5 sec and 8.1 sec. In this figure, the blue line is the situation of the processing of wait-for-memory timer. The upper side in the graph means the timer is on, i.e., the function is blocked by waiting for memory release. The lower side indicates the function is not blocked but working. The black points indicate that the penalizations are invoked.

This figure suggests the followings. First, the `tcp_sendmsg_locked()` function blocks very often in order to wait for the send buffer release, and the blocking keeps long and so the `tcp_sendmsg_locked()` function seems to be almost always blocked. During the blocked periods, the function cannot perform anything for resolving the throughput degradation. The other is that the penalization that reduces the congestion window size of the slow subflow does not occur so many times as the send socket buffer starvation. During the 0.6 sec shown in the figure, only two sets of penalization occur. It should be noted that, at each set of penalization, the actual window reduction occurs multiple times. In the case of this figure, three reductions happened at each penalization. Throughout a 10 sec data transfer analyzed here, the buffer starvation happened 278 times. On the other hand, the penalization occurred 28 times, each of which included one to three window reductions.

Based on those considerations, we can summarize the reason of the throughput degradation is the followings. First, the handling of send socket buffer starvation is done in the very beginning of TCP data send processing, and the way is just to block the control for some period. During this waiting period, no mechanisms to recover degraded throughput, such as the RP mechanism, can be invoked. Second, these mechanisms may be invoked when an MPTCP sender receives ACK segments, but the frequency of these invocations are much less than that of buffer starvations.

V. CONCLUSIONS

In this paper, we pointed up the MPTCP performance degradation in the situation that subflows have different transmission delays and the send socket buffer size is insufficient. We showed this situation by the experiments using the in-house network and discussed the details of the MPTCP parameters during the degradation. We also showed the internal structure of Linux MPTCP software focusing on the buffer starvation and the MPTCP scheduler. In the end, we showed a possible reason why the performance degradation occurs. We are going to propose a new scheduler function to resolve this degradation caused by the insufficient send socket buffer size.

REFERENCES

[1] A. Ford, C. Raiciu, M. Handley, S. Barre, and J. Iyengar, “Architectural Guidelines for Multipath TCP Development,” IETF RFC 6182, Mar. 2011.

[2] A. Ford, C. Raiciu, M. Handley, and O. Bonaventure, “TCP Extensions for Multipath Operation with Multiple Addresses,” IETF RFC 6824, Jan. 2013.

[3] C. Raiciu, M. Handley, and D. Wischik, “Coupled Congestion Control for Multipath Transport Protocols,” IETF RFC 6356, Oct. 2011.

[4] M. Scharf and S. Kiesel, “Head-of-Line Blocking in TCP and SCTP: Analysis and Measurements,” IEEE GLOBECOM ’06, pp.1-5, Nov. 2006.

[5] Icteam, “MultiPath TCP – Linux Kernel implementation, Users:: Confiugre MPTCP,” https://multipath-tcp.org/pmwiki.php/Users/ConfigureMPTCP.

[6] C. Raiciu, et al., “How Hard Can It Be? Designing and Implementing a Deployable Multipath TCP,” 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI ’12), pp.1-14, Apr. 2012.

[7] O. Paasch, S. Ferlin, O. Alay, and O. Bonaventure, “Experimental Evaluation of Multipath TCP Schedulers,” 2014 SIGCOMM Workshop on Capacity Sharing Workshop (CSWS ’14), pp.27-32, Aug. 2014.

[8] N. Kuhn, et al., “DAPS: Intelligent delay-aware packet scheduling for multipath transport," IEEE ICC 2014, pp. 1222-1227, Jun. 2014

[9] F. Yand, Q. Wang, and P. D. Amer, “Out-of-oder Transmission for In-order Arrival Scheduling for Multipath TCP,” 28th International Conference on Advanced Information Networking and Aplications Workshops, pp. 749-752, May 2014.

[10] S. Ferlin, O. Alay, O. Mehani, and R. Boreli, “BLEST: Blocking Estimation-based MPTCP Scheduler for Heterogeneous Networks,” IFIP Networking 2016, pp. 431-439, May 2016.

[11] C. Paasch, R, Khalili, and O. Bonaventure, “On the Benefits of Applying Experimental Design to Improve Multipath TCP,” 9th ACM Conference on Emerging Networking Experiments and Technologies (CoNEXT ’13), pp.393-398, Dec. 2013.

[12] B. Arzani, A. Gurney, S. Cheng, R. Guerin, and B. T. Loo, “Impact of Path Characteristics and Scheduling Policies on MPTCP Performance,” 28th International Conference on Advanced Information Networking and Applications Workshops, pp.743-748, May 2014.

[13] J. Kim, B. Oh, and J. Lee, “Receive Buffer based Path Management for MPTCP in Heterogeneous Networks,” 2017 IFIP/IEEE Symposium on Integrated Network and service Management (IM), pp.648-651, May 2017.

[14] F. Zhou, et al., “The Performance Impact of Buffer Sizes for Multi-Path TCP in InternetSetups. In: 2017 IEEE 31st International Conference on Advanced Information Networking and Applications (AINA), pp.9-16, Mar. 2017.

[15] P. Dong, et al., “Performance Evaluation of Multipath TCP Scheduling Algorithms,” IEEE Access, Vol. 7, pp. 29818-29825, Feb. 2019.

[16] T. Kato, M. Tenjin, R. Yamamoto, S. Ohzahata, and H. Shinbo, “Microscopic Approach for Experimental Analysis of Multipath TCP Throughput under Insufficient Send/Receive Socket Buffers.,” 15th International Conference WWW/Internet 2016, pp.191-199, Oct. 2016.

[17] T. Kato, A. Diwakar, R. Yamamoto, S. Ohzahata, and N. Suzuki, “How Insufficient Send Socket Buffer Affects MPTCP Performance over Paths with Different Delay,” 6th World Conference on Information Systems and Technologies (WorldCIST 18), pp. 614-624, Mar. 2018.

[18] N. Handley, J. Padhye, and S. Floyd, “TCP Congestion Window Validation,” IETF RFC 2861, Jun. 2000.