NON-RENEGABLE SELECTIVE ACKNOWLEDGMENTS AND SCHEDULING FOR TCP AND MULTIPATH TCP

by

Fan Yang

A dissertation submitted to the Faculty of the University of Delaware in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science

Spring 2015

NON-RENEGABLE SELECTIVE ACKNOWLEDGMENTS AND SCHEDULING FOR TCP AND MULTIPATH TCP

by

Fan Yang

Approved: ____

Errol L. Lloyd, Ph.D.
Chair of the Department of Computer and Information Sciences

Approved: ____

Babatunde A. Ogunnaike, Ph.D.
Dean of the College of Engineering

Approved: ____

James G. Richards, Ph.D.
Vice Provost for Graduate and Professional Education

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: ____

Paul D. Amer, Ph.D.
Professor in charge of dissertation

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: ____

Adarshpal S. Sethi, Ph.D.
Member of dissertation committee

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: ____

Michela Taufer, Ph.D.
Member of dissertation committee

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: ____

Stephan Bohacek, Ph.D.
Member of dissertation committee

ACKNOWLEDGEMENTS

I am the luckiest student for having Prof Paul D. Amer as my advisor. I would like to express my special appreciation and thanks to him for encouraging my research and for allowing me to grow as a researcher. With his patience and help, I felt very warm and went through the most difficult time in my life. His strict attitude towards both research and life influenced me so much, and I try to remember each sentence he said to me. These are the most precious wealth of my whole life. Since I am an international student, he always taught me to speak English slowly and clearly. Also, he taught me to present complex ideas in a simple and clear way in scientific writings. I would say he changed my whole life. Really thank you so much Prof Amer.

I would also like to thank my committee members, Prof Adarsh Sethi, Prof Michela Taufer and Prof Stephan Bohacek for reviews, constructive suggestions and criticisms that helped to improve this dissertation, for letting my defense be an enjoyable moment, and for the brilliant comments and suggestions.

During my years at University of Delaware, I worked with a lot of great people in the Protocol Engineering Lab (PEL). Jonathan T. Leighton, Qi Wang and Ayush Dusia always provided invaluable comments and helpful discussions when I was stuck on a problem, and they also helped me a lot to improve my speaking English. Jiefu Li contributed to this dissertation by running experiments. In addition, I am very grateful to Nasif Ekiz for his help during my first years.

This dissertation is dedicated to my mother, Shunwen Ge. Words cannot express how grateful I am to her for all of the sacrifices that she has made on my behalf. Without her endless support, this dream would never come true. Although, my father, Shouxin Yang, has gone, I know his spirit is always with me and hope he be always happy in the heaven.

A special thanks goes to my wife Yang Xin and my daughter Tianze Yang, hope both of you enjoy your lives.

I would also like to thank all of my best friends Yuanfang Chen, Wei Wang, Boyu Zhang, Xiaoran Wang and Hao Feng who made Newark more fun and enjoyable. Thank you all.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

We investigate two issues related to the transport layer and propose solutions to address these issues. All proposed solutions are implemented in the Linux kernel and evaluated with real network topologies.

First, we explore what performance gains can be obtained when a TCP or Multipath TCP (MPTCP) receiver guarantees never to discard received out-of-order PDUs from the receive buffer (i.e., never reneg). TCP is designed to tolerate reneging. This design has been challenged since (i) reneging rarely occurs in practice, and (ii) even when reneging does occur, it alone generally does not help the operating system resume normal operation when the system is starving for memory. In the current MPTCP standard, an MPTCP receiver cannot selectively acknowledge the reception of out-of-order PDUs to an MPTCP sender. We investigate how freeing received out-of-order PDUs from the send buffer by using Non-Renegable Selective Acknowledgments (NR-SACKs) can improve end-to-end performance. This improvement results when send buffer blocking occurs in both TCP and MPTCP. Preliminary results for TCP NR-SACKs show that (i) TCP data transfers with NR-SACKs never perform worse than those without NR-SACKs, and (ii) NR-SACKs can improve end-to-end throughput when send buffer blocking occurs. Under certain circumstances, we observe throughput increasing by using TCP NR-SACKs as much as 15%. Preliminary results for MPTCP NR-SACKs show that (i) MPTCP data transfers with NR-SACKs never perform worse than those without NR-SACKs, and (ii) NR-SACKs can improve end-to-end throughput in MPTCP when send buffer blocking occurs. Under certain circumstances, we observe throughput increasing by using MPTCP NR-SACKs as much as 38%.

Second, we explore potential application performance gains from two innovative scheduling policies for MPTCP. Whenever an MPTCP sender wants to send data, the scheduler needs to decide on which subflow to send each byte. We explain problems with the default scheduler used by Linux MPTCP, and propose the design of a scheduler based not only on a subflow's 'speed' but also the subflow's congestion. Preliminary results show that our proposed scheduler improves the throughput in MPTCP by alleviating the problems caused by the default scheduler. We also define and use one-way communication delay of a TCP connection to design an MPTCP scheduler that transmits PDUs out-of-order over different subflows such that their arrival is in-order. Preliminary results show our proposed scheduler can reduce receive buffer utilization, and increase throughput when a small receive buffer size results in receive buffer blocking.

Chapter 1

INTRODUCTION

1.1 Dissertation Scope

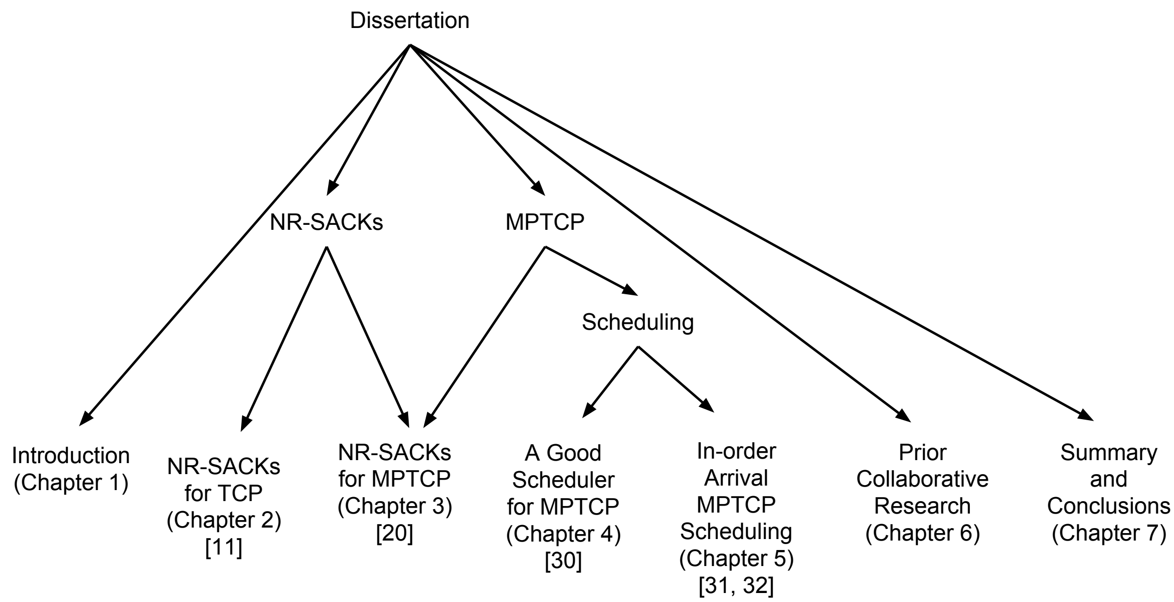

Figure 1.1: Dissertation Structure

This dissertation investigates two issues related to the transport layer: (i) potential application performance gains if TCP and Multipath TCP (abbreviated MPTCP), a transport layer protocol designed to concurrently use multiple TCP connections between multihomed hosts, do not tolerate reneging, and (ii) potential application performance gains from two different scheduling policies for MPTCP. The overall structure of the dissertation is shown in Figure 1.1.

TCP and MPTCP Non-Renegable Selective Acknowledgments (NR-SACKs) are analyzed in Chapters 2 and 3, respectively. Two MPTCP scheduling policies are described in Chapters 4 and 5, respectively. The references cited for each chapter represent the author's publications for each topic. Chapter 6 summarizes the author's collaborative work prior to the research contributions of this dissertation. Finally, Chapter 7 summarizes the author's contributions, and concludes this dissertation.

1.1.1 Reneging and NR-SACKs

Reliable transport protocols (such as TCP and SCTP) employ two kinds of data acknowledgment mechanisms: (i) cumulative acknowledgments (cumacks) indicate data that has been received in-sequence, and (ii) selective acknowledgments (SACKs) indicate data that has been received out-of-order. While cumacked data is a receiver's responsibility, SACKed data is not. SACKed out-of-order data is implicitly renegable; that is, a receiver may SACK data and later discard it [20]. The possibility of reneging forces a transport sender to maintain copies of SACKed data in the send buffer until they are cumacked.

TCP is designed to tolerate reneging. This design has been challenged [18] since (i) reneging rarely occurs in practice, and (ii) even when reneging does occur, it alone generally does not help the operating system resume normal operation when the system is starving for memory. If a TCP receiver never renegs, SACKed data is wastefully stored in the send buffer until cumacked.

Non-Renegable Selective Acknowledgments (NR-SACKs) were introduced in [17]. NR-SACKs allow a receiver to convey non-renegable information of received out-of-order data back to the corresponding sender. NR-SACKs allow that sender to remove NR-SACKed data from the send buffer sooner than waiting for the arrival of corresponding cumacks. NR-SACKs have been evaluated for both SCTP, and SCTP with Concurrent Multipath Transmission (CMT), and results show NR-SACKs not only reduce sender's memory requirements, but also improve the end-to-end throughput under certain conditions [14, 15, 19, 22]. In Chapter 2, this dissertation investigates potential application performance gains if a TCP receiver never renegs and likewise uses NR-SACKs.

1.1.2 Multipath TCP

A host is multihomed if it can be addressed by multiple IP addresses. Multihoming has increased the interest in using multiple paths simultaneously (i.e., CMT) for achieving higher reliable, end-to-end throughput, and increasing robustness during time of path failure.

Multipath reliable data transfer has received a lot of recent attention as seen by extensions to TCP and SCTP to support multihoming. However, the multihoming extensions to TCP [39, 40, 41] have never been implemented nor deployed [5]. SCTP with CMT is implemented but not widely deployed since many Internet middle-boxes by default block SCTP-PDUs.

To migrate multipath data transfer from theory to practice, the IETF has created a working group to specify a standard for Multipath TCP (MPTCP). MPTCP, perhaps the most significant change to TCP in the past 20 years [6], simultaneously transfers data on multiple TCP connections (subflows) between peers [1].

In the current MPTCP Linux implementation [35], an MPTCP receiver never renegs on received out-of-order MPTCP-PDUs. In Chapter 3, this dissertation introduces NR-SACKs to MPTCP, and investigates potential application performance gains. We extended the Linux MPTCP implementation to support NR-SACKs. Preliminary results show that (i) MPTCP data transfers with NR-SACKs never perform worse than those without NR-SACKs, and (ii) NR-SACKs can improve end-to-end throughput in MPTCP when send buffer blocking occurs.

An important component of MPTCP is the scheduler. Whenever an MPTCP sender wants to send data, the scheduler needs to decide on which subflow to send each byte (Figure 1.2). During Chapter 3 experiments on MPTCP NR-SACKs, we found a problem of the default scheduler of the Linux MPTCP. In Chapters 4 and 5, this dissertation investigates two different scheduling policies for MPTCP, and addresses these two scheduling policies to improve application performance. Chapter 4 explains problems with the default scheduler used by Linux MPTCP, and proposes the design of a scheduler which based on not only a subflow's 'speed' but also the subflow's congestion. Preliminary empirical results show that our proposed scheduler improves the throughput in MPTCP by alleviating the problems caused by the default scheduler.

Chapter 5 uses one-way communication delay of a TCP connection to design an MPTCP scheduler that transmits data out-of-order over multiple paths such that their arrival is in-order. Our Linux implementation shows our proposed scheduler can reduce receive buffer utilization, and increase throughput when a small receive buffer size results in receive buffer blocking.

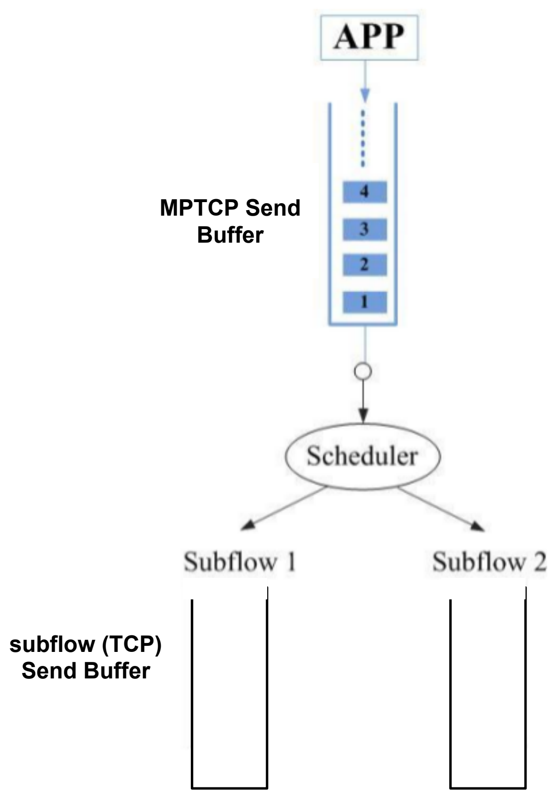

Figure 1.2: MPTCP Scheduler

1.2 MPTCP Primer

1.2.1 MPTCP in the Networking Stack

MPTCP operates at the upper part of the transport layer, and aims to be transparent to both higher and lower layers [1]. MPTCP provides a set of additional features on top of standard TCP. The layering is shown in Figure 1.3.

```
+-------------------------------+
|           Application         |
+-------------------------------+
|             MPTCP             |
+ - - - - - - - + - - + - - - - - - - +
| Subflow (TCP) | ... | Subflow (TCP) |
+---------------+-----+---------------+
|       IP      | ... |       IP      |
+---------------+-----+---------------+
```

Figure 1.3: MPTCP in the Networking Stack

1.2.2 MPTCP Connection Establishment

The connection establishment of the first subflow is same as that of a TCP connection, but the SYN, SYN/ACK, and ACK TCP-PDUs carry a new MP_CAPABLE option. This MP_CAPABLE option verifies whether both end hosts support MPTCP. After the first subflow is established, additional subflows can be established, and the SYN, SYN/ACK, and ACK TCP-PDUs contain a new MP_JOIN option. Figure 1.4 shows the establishment of an MPTCP connection with two subflows between hosts A and B. Host A is multihomed with two interfaces A_1 and A_2, and host B has one interface B.

1.2.3 Data Transfer Using MPTCP

In MPTCP, each subflow is a standard TCP connection with its own sequence number space. An MPTCP level sequence number called the Data Sequence Number (DSN) additionally numbers bytes at the MPTCP level. A single MPTCP send buffer and a single MPTCP receive buffer are shared among all subflows, while each subflow

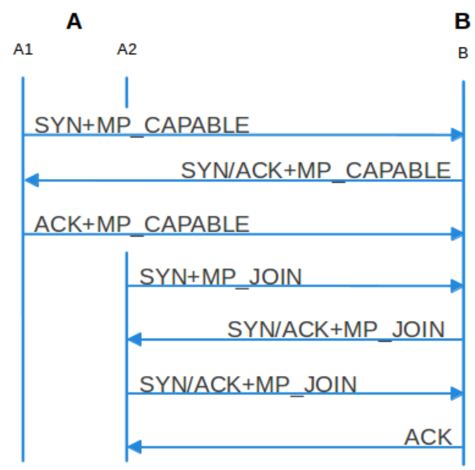

Figure 1.4: MPTCP Connection Establishment

has its own receive buffer to hold subflow level out-of-order data (since each subflow TCP receiver must deliver subflow level data in-order to the MPTCP receive buffer).

When an application writes a stream of bytes to an MPTCP send buffer, MPTCP numbers each byte with a DSN. Then a scheduler runs to select which subflow(s) to send the data. Bytes are then transmitted on the selected subflow(s) where they are encapsulated into TCP-PDUs with MPTCP information placed in the TCP option field. When a TCP-PDU is received in-order at the subflow level, the payload is delivered to the MPTCP receive buffer immediately. The MPTCP level cumack number, called DATA ACK (DA), advances if the delivered data are also in-order at the MPTCP level.

The subflow receiver cumacks those delivered data using a regular TCP cumack, and places the current DA in the TCP option field. An application consumes in-order data from the MPTCP receive buffer. Currently, an MPTCP sender only frees data from the MPTCP send buffer when they have been cumacked by DA received on any subflow.

Figure 1.5 shows an example of data transfer using MPTCP with two subflows.

Each data PDU contains 1400 bytes of data, and is represented by an arrow with both subflow sequence number (Seq) and DSN of the first byte. Each ACK PDU is represented by an arrow with both subflow ACK number (Ack) and DA.

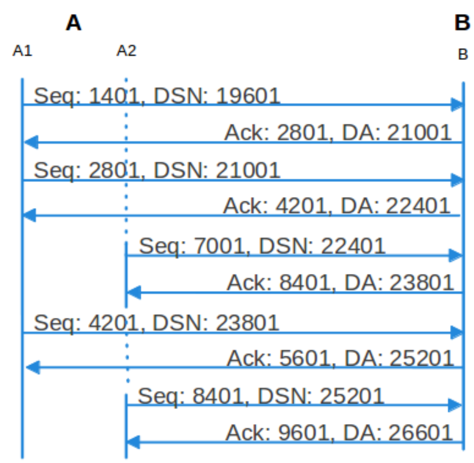

Figure 1.5: Data Transfer Using MPTCP

1.2.4 MPTCP Connection Termination

When host B wants to inform host A about the end of data transfer, host B sends a 'DATA FIN' (which has the same semantics and behavior as a regular TCP FIN) at the MPTCP level. Once all the data on the MPTCP connection has been successfully received, all subflows close in the same manner as a regular TCP connection.

Chapter 2

NON-RENEGABLE SELECTIVE ACKNOWLEDGMENTS (NR-SACKS) FOR TCP

In TCP, Selectively Acknowledged (SACKed) out-of-order data is implicitly renegable; that is, the receiver can SACK data and later discard it [20]. The possibility of this reneging forces the sender to maintain copies of SACKed data in the send buffer until a later time when the data are cumulatively ACKed. Based on prior research concluding that TCP's tolerance of reneging is inefficient [18], we investigate what performance gains can be obtained by assuming reneging by a TCP receiver is not permitted, thus allowing a TCP sender to immediately free SACKed data from its send buffer. The difficulty of implementing NR-SACKs in the Linux kernel was far beyond our initial expectation. The TCP code, roughly 100K lines of C code, in the Linux kernel keeps improving as the version changes. During the process to understand the TCP implementation thoroughly, we tried several different ways to add NR-SACKs.

2.1 Reneging

TCP uses sequence numbers and cumulative acknowledgments to achieve reliable data transfer. A TCP data receiver uses sequence numbers to sort arrived data segments. Data arriving in expected order, i.e., ordered data, results in a cumulative ACK (cumack) being transmitted back to the data sender. A cumack semantically means the data receiver accepts full responsibility of delivering the data to the receiving application. Relieved of this responsibility, the data sender therefore deletes all cumacked data from its send buffer, possibly even before that data gets delivered from the TCP receiver's buffer to the appropriate receiving application.

The receive buffer consists of two types of data: ordered data which has been cumacked but not yet delivered to the application, and out-of-order data that resulted from loss or reordering in the network. A TCP data receiver must not delete cumacked data without delivering it since the data sender will have removed cumacked data from its send buffer. That is, a receiver must not reneg on cumacked data.

The Selective Acknowledgment Option (SACK) [20] is an extension to TCP's cumulative ACK mechanism, and is used by a data receiver to selectively acknowledge arrived out-of-order data to the data sender. The intent is that SACKed data do not need to be retransmitted by the sender during loss recovery.

Data receiver reneging (or simply *reneging*) occurs when a data receiver SACKs data, and afterwards discards that data from its receive buffer without delivering the data to the receiving application or socket buffer. TCP is designed to tolerate data reneging. Specifically [20] states: "The SACK option is *advisory*, in that, while it notifies the data sender that the data receiver has received the indicated segments, the data receiver is permitted to later discard data which have been reported in a SACK option". Therefore, a TCP data sender must retain copies of all transmitted data in its send buffer, even SACKed data, until they are cumacked.

The design of tolerating data reneging in TCP has been challenged [18] since (i) reneging rarely occurs in practice, and (ii) even when it does occur, reneging alone generally does not help the operating system resume normal operation when the system is starving for memory. Based on this conclusion, SACKed data is wastefully stored in the send buffer until cumacked. We consider the potential performance gains for TCP if its design were not to tolerate reneging.

2.2 Potential Performance Gains by Prohibiting Reneging in TCP

To gain insight to the performance penalty incurred by TCP tolerating reneging, consider the example in Figure 2.1. Assume the shown TCP send buffer can accommodate four TCP-PDUs and the TCP receive buffer can hold seven TCP-PDUs. As the TCP sender transmits TCP-PDUs, space is allocated in the send buffer. When cumacks come back to the sender, the cumacked data is released. When SACKs come back, information is noted at the data sender, but the data itself cannot be released. Only later when SACKed data is eventually cumacked will the allocated send buffer space be released. During the intervals between SACKing and cumacking, the send buffer utilization falls below 100%. For example in Figure 2.1, after the "ACK 1, SACK 3-4" arrives, half of the send buffer is storing data that has already arrived at the data receiver. If the send buffer is small as in this illustration, a situation arrives after TCP-PDU 5 is transmitted when no new data can be transmitted until TCP-PDU 2 is retransmitted and later cumacked. This situation is referred to as *send buffer blocking*.

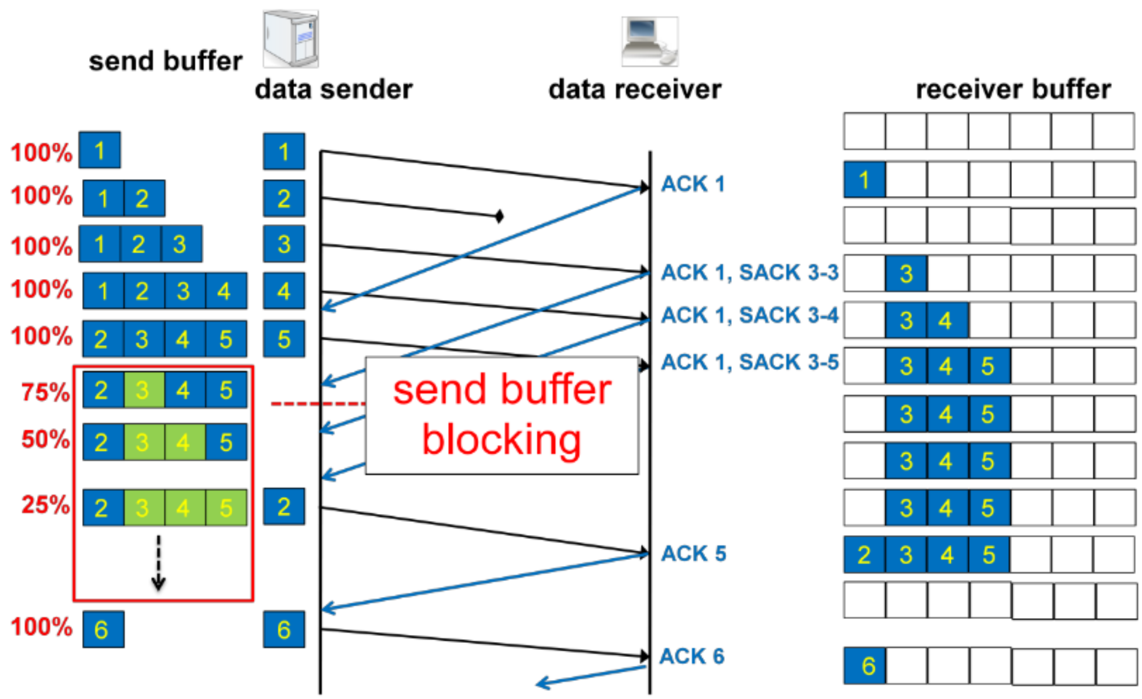

Figure 2.1: TCP Data Transfer: Normal

Figure 2.2 illustrates the potential performance gain if the SACKed data were non-renegable, and thus could be removed as would be the case when reneging is forbidden. The TCP sender is not blocked "wastefully" maintaining copies of SACKed data. Instead the send buffer has room for transmitting new application data.

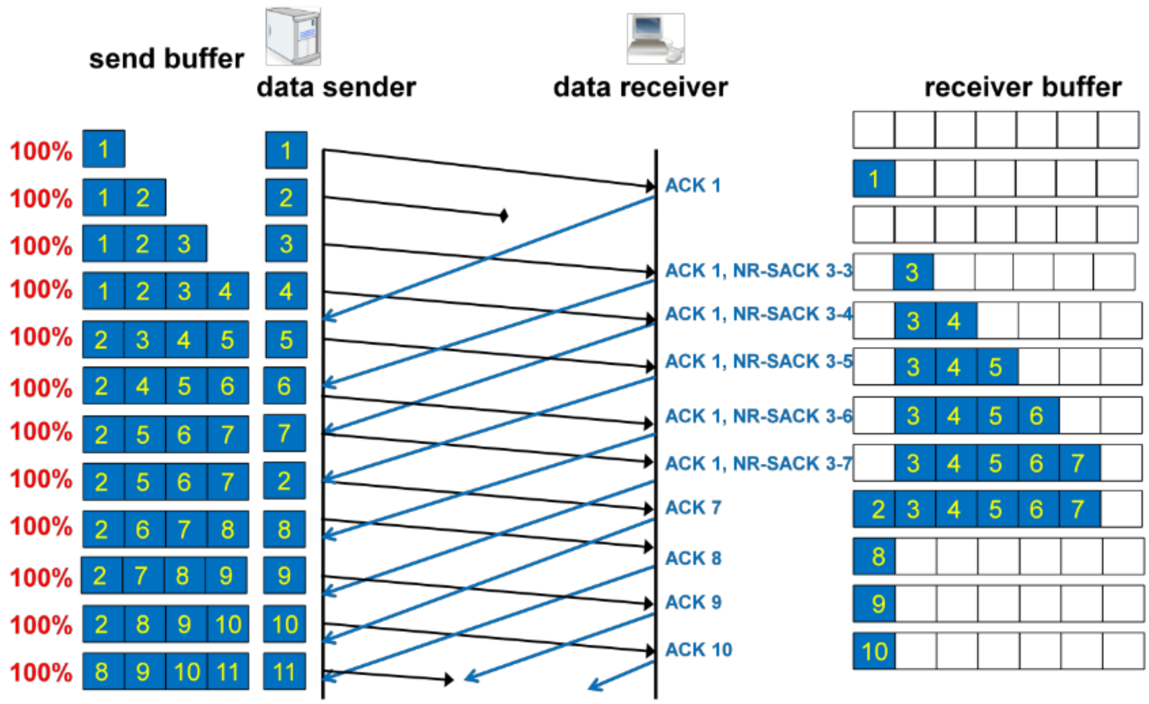

Figure 2.2: TCP Data Transfer: NR-SACKs

2.3 Discussion

First, unlike SCTP's unordered data service which allows a data receiver to deliver out-of-order data to a receiving application, a TCP receiver must not deliver out-of-order data. Whereas SCTP's receiver effectively advertises extra available receive window space upon delivering out-of-order data, TCP's receiver must keep the out-of-order data and does not increase the receiver window size.

Second, the current semantics of a TCP send buffer define a window of contiguous bytes that a sender may transmit. The lower edge of the window is defined by the received highest cumack number. The upper edge is defined to be the highest cumack number plus the number of bytes in the advertised receive window.

Under these two circumstances, there is no advantage to having a receive window larger than the send window (as demonstrated in Figure 2.1). We propose to modify the TCP's send window semantics to **allow a possibly non-contiguous set of bytes**. Please note, the advertised receive window semantics does not change; it remains the number of bytes that the data sender is allowed to have outstanding starting from the received highest cumack number. With this change, the send buffer may have gaps. For example, in Figure 2.2, after TCP-PDU 3 is freed by NR-SACK 3-3, the send buffer is not contiguous. Now, it makes sense to have a receiver window larger than the send window. A smaller send buffer, which needs not to keep copies of SACKed data, can keep a larger receive window busy (e.g., default send and receive buffer sizes for Linux 2.6.31 are 16,384 and 87,380 bytes, respectively.)

2.4 Implementation

This section describes the complexities incurred in implementing TCP NR-SACKs in the Linux kernel. Important to note: **NR-SACK changes only modify the sender; the receiver structures are unchanged**. Coding in Linux kernel is challenging; the key is thoroughly understanding critical data structures. We start this section by introducing two core data structures.

2.4.1 Critical Data Structure I: `sk_buff` Structure

The `sk_buff` structure (skb) is a central data structure in the Linux networking code, representing data that is about to be transmitted by a sender or has been received by a receiver. An skb comprises three elements:

- an sk_buff structure which contains control information
- linear data (introduced in section 2.4.1.1.)
- nonlinear data (introduced in section 2.4.1.1.)

Figure 2.3 shows the fields related to NR-SACKs implementation in the sk_buff structure:

```c
struct sk_buff {
    struct sk_buff *next;
    struct sk_buff *prev;
    //...
    unsigned int len;
    unsigned int data_len;
    unsigned int truesize;
    //...
    char cb[48];
    //...
    unsigned char *head;
    unsigned char *data;
    unsigned char *tail;
    unsigned char *end;
    //...
};
```

Figure 2.3: sk_buff Structure

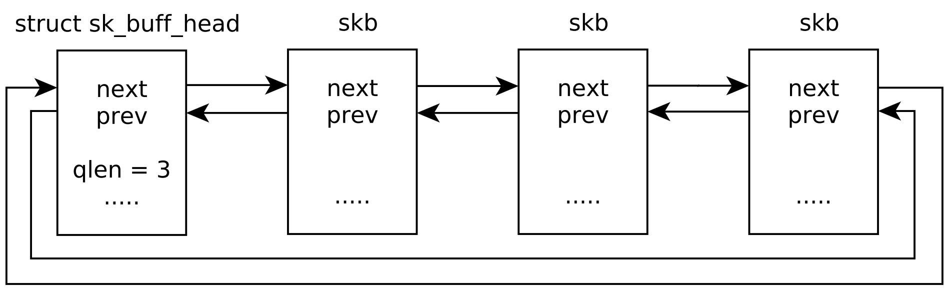

Figure 2.4: A List of skbs

next and prev: two fields link different skbs together. The kernel maintains skbs in a doubly linked list (Figure 2.4). The `sk_buff_head` structure represents the head of a list, and the *qlen* field indicates the number of skbs in the list. In TCP, both send and receive buffers are represented by lists of skbs.

len: indicates the total data (includes both linear and nonlinear data sections) length of this skb.

data_len: indicates the nonlinear data length of this skb. Obviously, the linear data length is $len - data\_len$.

true_size: indicates the total size of this skb, including both the sk_buff structure and the data sections.

cb: (introduced in section 2.4.1.2.)

head: points to the start of the linear data section.

data: points to the payload of the linear data section. Protocol headers reside between head and data pointers.

tail: points to the end of the payload. Some protocol control information (e.g., Ethernet checksum) is later placed after the actual payload.

end: points to the end of the linear data section. Here, end pointer also points to the start of the nonlinear data section.

Nonlinear data section is represented by the `skb_shared_info` structure. `skb_shared_info` contains a list of `skb_frag_t` and each `skb_frag_t` points to a memory block inside a memory page.

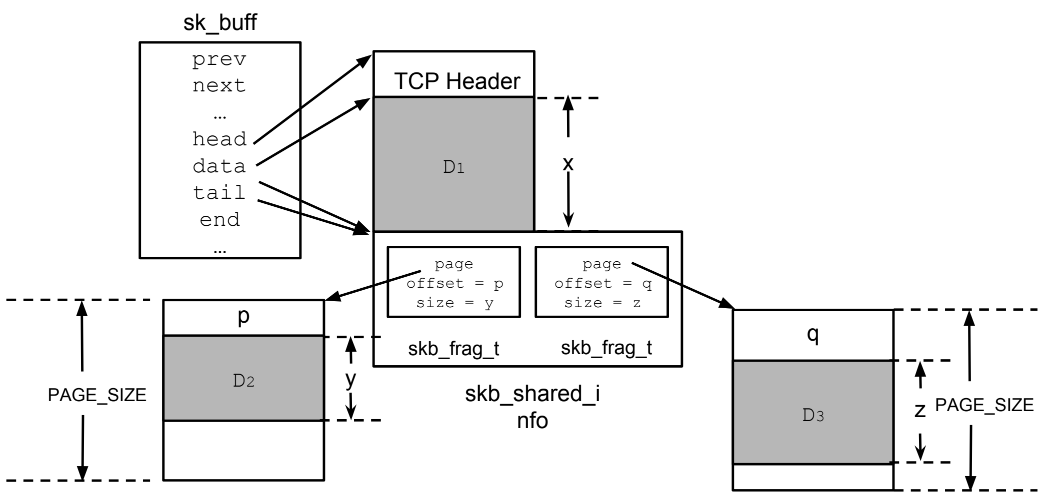

Figure 2.5: A TCP-PDU in Linux Kernel

Figure 2.5 shows a TCP-PDU in the Linux kernel. The application layer data is represented by shaded boxes D_1, D_2 and D_3. The linear data section contains a payload (D_1) of size x. The nonlinear data section (represented by a `skb_shared_info` structure) contains two `skb_frag_t`s. The payload (D_2 and D_3) sizes of these two `skb_frag_t` are y and z, respectively. Thus, in this example, $len = x + y + z +$

size of the TCP Header, $data\_len = y + z$, and $true\_size$ includes all memory allocated for this TCP-PDU.

During transmission of a PDU (an skb passes down the protocol stack layers), the header of each layer is added into the space between *head* and *data*. At the receiver, on receipt of a PDU (an skb passes up the protocol stack layers), the header of each lower layer is removed. In this dissertation, The words 'skb' and 'PDU' are interchangeable, since skb is the data structure which represents PDUs in Linux. Note: an skb can represent multiple PDUs.

2.4.1.1 Memory Allocation for an skb

Memory allocation is important for our implementation, since the key part of NR-SACKs is memory manipulation.

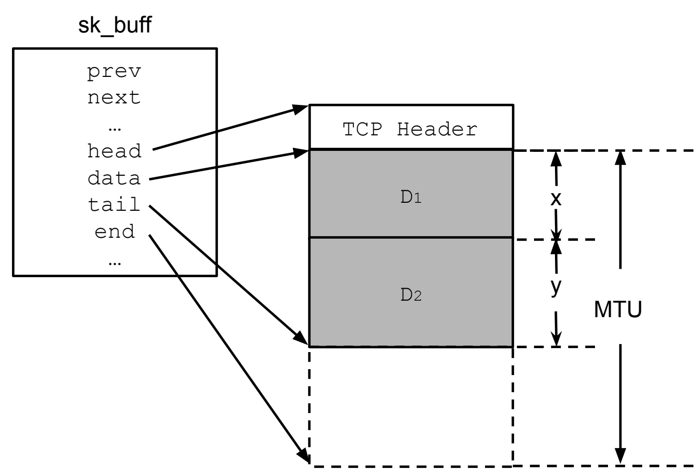

Figure 2.6: Allocation of skb without Scatter/Gather I/O

First, let us see why an skb needs both the linear and nonlinear data sections. Consider an example: the application of a TCP connection generates two small data chunks (denoted as D_1 and D_2) of size x and y ($x + y \leq$ MSS (Maximum Segment Size)), respectively. A TCP sender has two options to allocate an skb: (i) allocate a

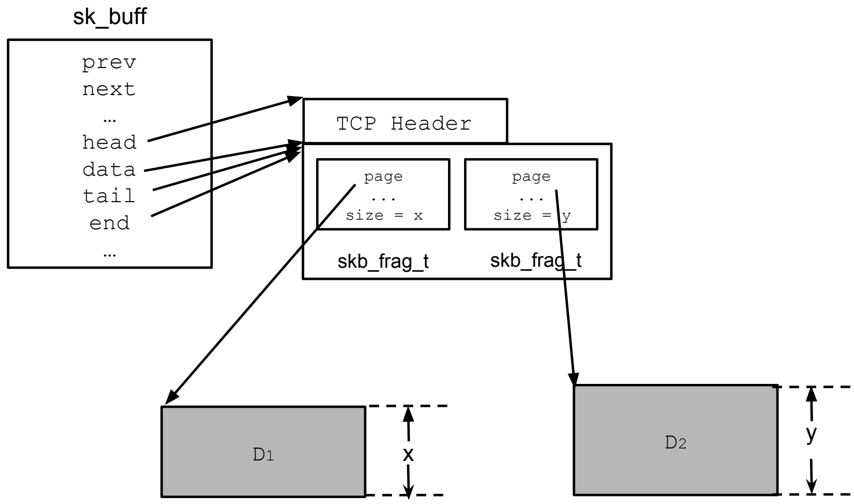

Figure 2.7: Allocation of skb with Scatter/Gather I/O

linear data section of size MTU (Maximum Transmission Unit) and copy both data chunks to the linear data section (Figure 2.6), or (ii) allocate a linear data section just to hold protocol headers and make pointers in the nonlinear data section point to both data chunks (Figure 2.7). The second choice is more efficient because less memory copies are involved, but needs a support, called Scatter/Gather I/O, from the network interfaces (e.g., the Ethernet interface). An interface, which supports Scatter/Gather I/O, can gather these physically non-continuous data chunks and transmit the chunks in one PDU. Nowadays, almost all network interfaces support Scatter/Gather I/O, so an skb can use nonlinear data section to avoid memory copies.

2.4.1.2 Control Buffer Field

The `sk_buff` structure contains a field (*cb*), called a 'control buffer', which is used by each layer to store internal control information. The information in this field changes as the skb traverses different layers. For example, when an skb is in the TCP send buffer, the skb's *cb* stores a `tcp_skb_cb` structure which contains information such as: start and end sequence numbers, time when this skb is sent, etc (Figure 2.8).

```
struct tcp_skb_cb {
    //...
    __u32 seq;       /* Starting sequence number */
    __u32 end_seq;   /* SEQ + FIN + SYN + datalen */
    __u32 when;      /* used to compute rtt's */
    //...
    __u8 sacked;     /* State flags for SACK/FACK. */
    #define TCPCB_SACKED_ACKED      0x01    /* SKB ACK'd by a SACK block */
    #define TCPCB_SACKED_RETRANS    0x02    /* SKB retransmitted */
    #define TCPCB_LOST              0x04    /* SKB is lost */
    #define TCPCB_EVER_RETRANS      0x80    /* Ever retransmitted frame */
    #define TCPCB_RETRANS           (TCPCB_SACKED_RETRANS|TCPCB_EVER_RETRANS)
    //...
};
```

Figure 2.8: tcp_skb_cb Structure

An important field in the `tcp_skb_cb` structure is *sacked* (8-bit), which is used to record the state of an skb in the send buffer:

`TCPCB_SACKED_ACKED`: indicates the skb has been SACKed if the first bit is 1.

`TCPCB_SACKED_RETRANS`: indicates the skb has been retransmitted after being SACKed if the second bit is 1.

`TCPCB_LOST`: indicates the skb is presumed to be lost if the third bit is 1.

`TCPCB_EVER_RETRANS`: indicates the skb has been retransmitted after being presumed to be lost if the eighth bit is 1.

The *sacked* of each skb is updated when an acknowledgment comes back. Based on the states of all skbs in the send buffer, a TCP sender estimates the current state of network, and the congestion control mechanism determines to increase or slow down transmission.

2.4.2 Critical Data Structure II: `tcp_sock` Structure

The `tcp_sock` structure (Figure 2.9) describes a TCP connection. Since NR-SACKs do not change any fields with receiver side information, this section focuses on the fields which contain sender side information.

```c
struct tcp_sock {
    //...
    __u32 snd_una;      /* First byte we want an ack for */
    __u32 snd_nxt;      /* Next sequence we send */
    //...
    __u32 packets_out;  /* Packets in the retransmission queue */
    __u32 sacked_out;   /* SACK'd packets */
    __u32 lost_out;     /* Lost packets */
    __u32 fackets_out;  /* FACK'd packets */
    __u32 retrans_out;  /* Retransmitted packets out */
    //...
};
```

Figure 2.9: tcp_sock Structure

The left edge of a TCP send buffer is denoted as *snd_una* which is the first byte the sender wants an ack for. *snd_nxt* is the right edge of the retransmission queue and denotes the first sequence number which has not been sent yet.

As stated in section 2.4.1.2, each skb in the TCP retransmission queue is tagged by a *sacked* field. Based on the state of each skb, a TCP sender maintains per-socket information to estimate current network capacity. This estimate is used by both congestion control and flow control mechanisms. Using NR-SACKs does not modify congestion control or flow control mechanisms, so only the fields related to NR-SACKs are introduced:

packets_out: number of TCP-PDUs in the retransmission queue.

sacked_out: number of TCP-PDUs which have been SACKed. These SACKed TCP-PDUs are tagged as `TCPCB_SACKED_ACKED` in *sacked*.

lost_out: number of TCP-PDUs which are presumed to be lost. These TCP-PDUs are tagged as `TCPCB_LOST` in *sacked*.

fackets_out: number of TCP-PDUs which are forward acknowledged (FACKed) [9].

retrans_out: number of TCP-PDUs which have been retransmitted. These TCP-PDUs are tagged as either `TCPCB_SACKED_RETRANS` or `TCPCB_EVER_RETRANS` in *sacked*.

As stated in [9], $lost\_out = fackets\_out - sacked\_out$. Figure 2.10 demonstrates the relations of these fields by an example. In the figure, skbs 3 and 5 have been

SACKed, and skbs 7 and 8 have not been sent yet. The retransmission queue contains skbs 1 to 6, so $packets\_out = 6$. Two skbs have been SACKed, so $sacked\_out = 2$. SACKed skb with the highest sequence number is skb 5, so $fackets\_out = 5$ and $lost\_out = 5 - 2 = 3$.

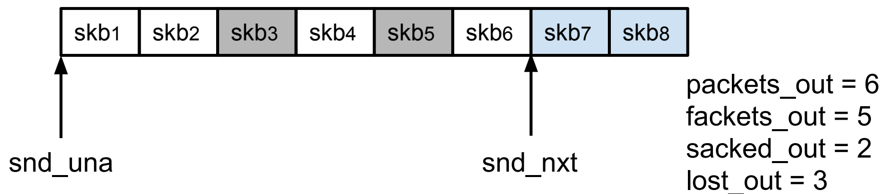

Figure 2.10: An Example TCP Send Queue

2.4.3 Processing of Incoming NR-SACKs

Implementation of NR-SACKs on the TCP receiver side is trivial. Reneging is turned off by commenting out `tcp_collapse_ofo_queue()` in `net/ipv4/tcp_input.c`. Then, all SACKs can be treated as NR-SACKs.

The TCP sender processes incoming acks in `tcp_ack()`. If the incoming ack contains SACKs, these SACKs may update the states of skbs in the send buffer and corresponding fields in the `tcp_sock` will be updated. For the example in Figure 2.10, if skb 4 is reported by SACKs, then skb 4 is tagged as `TCPCB_SACKED_ACKED` in the *sacked*. Correspondingly, *sacked_out* and *lost_out* are updated to 3 and 2, respectively. To save memory space, adjacent skbs are combined if they have all been SACKed. Therefore, skbs 3 to 5 will be combined to one skb after skb 4 is SACKed.

Now, the problem seems to be simple. Just deallocate the memory occupied by skbs tagged as `TCPCB_SACKED_ACKED`, and we are done. This is exactly what we did at the beginning. Then we tested this modified TCP with a file transfer. Unexpectedly, the throughput by using this modified TCP was always lower than that by using the regular TCP. The reason is: when new SACKs are received, the TCP sender updates *fackets_out*, *sacked_out* and *lost_out*. But after the SACKed skbs are deallocated, the *sacked_out* is updated to 0 and the $lost\_out = fackets\_out$. The sender infers that all TCP-PDUs which have not been cumacked are lost, which means the network is so congested that it just drops rather than reorders TCP-PDUs. As a result, the sender slows down the transmission, and the throughput decreases.

After this first attempt, our thought was: the problem came from not correctly updating *fackets_out* and *lost_out* after SACKed skbs are deallocated. That is, since all SACKed skbs are deallocated, both *fackets_out* and *lost_out* must be 0. Similarly, if the SACKed skbs has been retransmitted (tagged as `TCPCB_SACKED_RETRANS`), *retrans_out* needs to be updated also. However, when we tested this modified version, the throughput became even worse. The reason is: since the *fackets_out*, *sacked_out* and *lost_out* are always 0, the sender infers that the network is good and all TCP-PDUs always arrive in-order at the receiver. As a result, the sender keeps increasing the cwnd and more losses occur.

By analyzing these two initial approaches, we see that the TCP senders in both initial approaches had **wrong estimates of the current state of the network**. These wrong estimates mislead the congestion control mechanism. The TCP sender in the first attempt under-estimates network capacity and unnecessarily throttles transmission. The TCP sender in the second implementation over-estimates network capacity and over-sends TCP-PDUs. If NR-SACKs only free data sections of a SACKed skb but maintain the `skb_buff`, corresponding fields (e.g., *fackets_out*, *sacked_out* and *lost_out*) in `tcp_sock` are same as normal TCP (without NR-SACKs). A TCP sender can have a correct estimate of the network state based on these fields. Since all information in both `sk_buff`s and `tcp_sock` remains unchanged but the data are freed, ancillary data structures are needed to manage this mismatch and keep track of data which have already been freed by NR-SACKs.

We introduce a structure `nrsack_block` which comprises a *start_seq* and a *end_seq* (Figure 2.11), indicating the data chunk (seq: *start_seq* (inclusive) to *end_seq* (not inclusive)) has been reported and deallocated by NR-SACKs. Each TCP sender maintains a `nrsack_list` which is a doubly linked list of `nrsack_blocks` (Figure 2.12). A `list_head` structure is a provided standard implementation of circular, doubly linked lists in the Linux kernel.

```
struct nrsack_block {
    struct list_head list;
    unsigned int start_seq;
    unsigned int end_seq;
};
```

Figure 2.11: nrsack_block Structure

```
struct tcp_sock {
    //...
    struct list_head nrsack_list;
    //...
};
```

Figure 2.12: nrsack_list Structure

2.4.4 Complexity Analysis of Implementation

Figures 2.13 shows the ultimate processing procedure of incoming NR-SACKs. In `tcp_sacktag_write_queue()`, all SACKed skbs are tagged as `TCPCB_SACK_ACKED` and adjacent SACKed skbs are combined. Note that, an skb can be partially SACKed. To deallocate the partially SACKed data part, the skb needs to be split into multiple skbs. For example, if skb (seq: 12001 - 15001) is partially SACKed by SACK block 13001 - 14001, the original skb will be split into three skbs: skb_1 (seq: 12001 - 13001), skb_2 (seq: 13001 - 14001) and skb_3 (seq: 14001 - 15001). Then only skb_2 is tagged as `TCPCB_SACK_ACKED`. Splitting is a reverse process of the combination operation and has constant time cost.

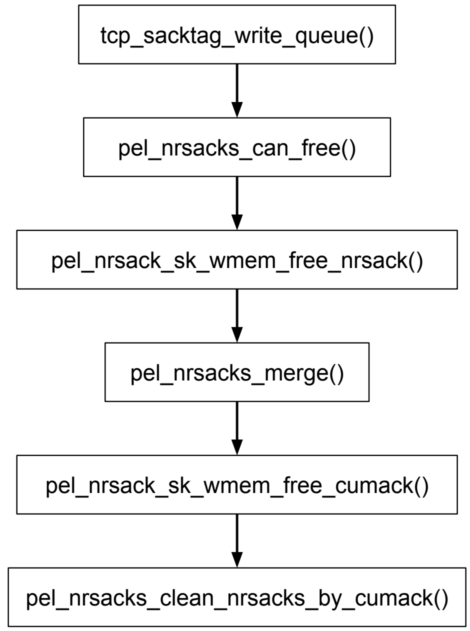

Figure 2.13: NR-SACKs Processing

In `pel_nrsacks_can_free()`, for each skb in the retransmission queue, the number of bytes freed by a newly received NR-SACKs (`bytes_freed_by_nrsacks`) is calculated, and the corresponding memory is freed. Note that, although the memory is deallocated, the information in the `sk_buff` remains unchanged. Thus, for a newly received SACK block, all existing `nrsack_blocks` in the `nrsack_list` need to be examined to determine which bytes already have been freed. Here, although the sequence numbers in the `sk_buff` are contiguous, the **actual data in the send buffer are not contiguous**. The current `nrsack_list` shows the gaps in the send buffer. For example, assume `nrsack_list` contains two `nrsack_blocks`: 2000 - 3000 and 4000 - 5000, and the current `snd_una` is 1000. A newly incoming NR-SACK block is 3500 - 4500. Figure 2.14 shows part of the retransmission queue and the `nrsack_list`. For

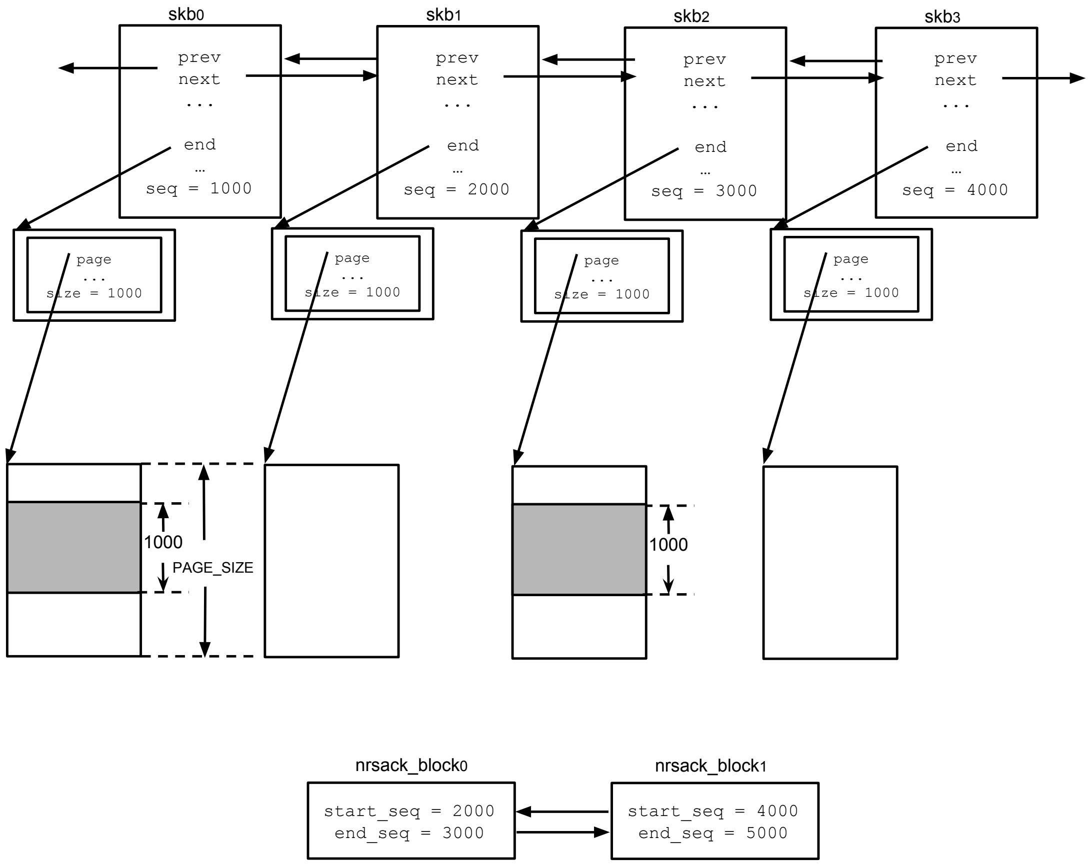

Figure 2.14: Before Processing NR-SACK Block 3500 - 4500

the purpose of simplicity, TCP headers in the linear data sections are not shown. Although the data of skb_1 and skb_3 has been deallocated by NR-SACKs, the information in both `sk_buff`s remains unchanged. The number of additional bytes freed by 3500 - 4500 is 500 bytes (seq: 3500 - 4000) since the data (seq: 4000 - 4500) has already been freed. After this new NR-SACK block has been processed, part of the retransmission queue is shown in Figure 2.15. All `sk_buff`s remain unchanged, and the actual data of skb_2 reduces to 500 bytes. The send buffer has two gaps 2000 - 3000 and 3500 - 5000. Assume **n** existing `nrsack_block`s, and **m** skbs in the send buffer, the worst case running time to process one newly arrived NR-SACK block is $O(m + n)$.

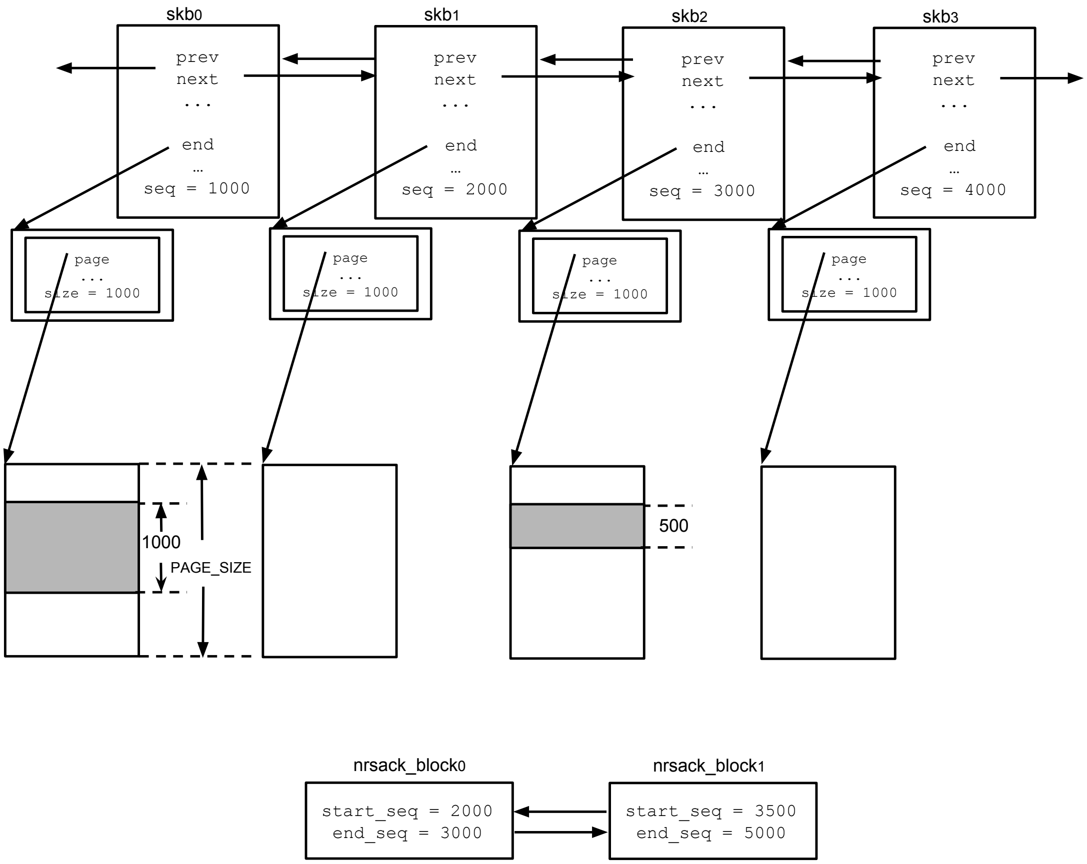

Figure 2.15: After Processing NR-SACK Block 3500 - 4500

`pel_nrsack_sk_wmem_free_nrsack()` updates the current send queue size and available memory space. For the above example, the send queue size is decreased by 500 bytes, and the available memory for the send queue is increased by 500 bytes. The running time of this function is O(1).

In `pel_nrsacks_merge()`, newly received NR-SACK block are added to `nrsack_list` and the block is merged with existing `nrsack_blocks` if possible. For the example in Figure 2.14, 3500 - 4500 would be merged with 2000 - 3000 and 4000 - 5000, and `nrsack_list` would contain two `nrsack_blocks`: 2000 - 3000 and 3500 - 5000 (Figure 2.15). The worst case running time of this function is O(n). Merging `nrsack_blocks` can decrease the number of `nrsack_block`s in the `nrsack_list`, thus improving the efficiency of other NR-SACK processing functions.

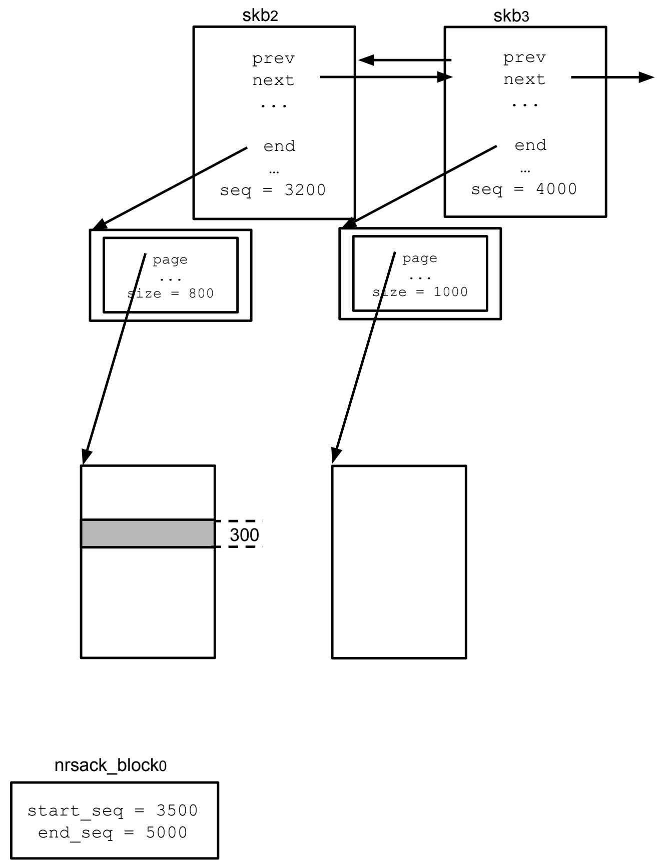

Figure 2.16: After Receiving Cumack 3200

In `pel_nrsack_sk_wmem_free_cumack()`, the number of bytes freed by cumacks is calculated and the corresponding memory is freed. Note that, the `sk_buff` is also deallocated here. Similarly, all of the existing `nrsack_block`s in the `nrsack_list` need to be examined to determine which bytes have already been freed. Also, the send queue size and the available memory for the send queue are updated. For the example in Figure 2.15, assume a cumack = 3200 is received, only 1200 bytes (sequence numbers 1000 - 2000 and 3000 - 3200) plus the size of the `sk_buff`s of skbs 0 and 1 can be freed by this cumack since the sequence space 2000 - 3000 has already been freed. In Figure 2.16, both skb_0 and skb_1 are deallocated. Note: now skb_3 has a start sequence number $= 3200$ and length of data $= 800$, but the actual data length is 300 bytes. The worst case running time of this function is also O(n).

In `pel_nrsacks_clean_nrsacks_by_cumack()`, some NR-SACK blocks in the `nrsack_list` is freed by cumacks, since the blocks which are under the cumack are unneeded. Use the above example, after cumack $= 3200$ is received, the `nrsack_list` contains only one `nrsack_block`: 3500 - 5000. The worst case running time of this function is O(m).

A possible improvement is to add shortcuts. We can observe in TCP transmission, when a TCP-PDU is lost, the TCP-PDUs after the lost one can arrive at the receiver and are reported by NR-SACKs in sequential order. In `pel_nrsacks_can_free()`, for a newly incoming NR-SACK block $(\mathrm{S_{new}} - \mathrm{E_{new}})$, the TCP sender can first determine whether $\mathrm{S_{new}} \geq$ the end sequence $(\mathrm{E_{end}})$ of the last `nrsack_block` in current `nrsack_list`. If the answer is yes (shortcut hit), the sender does not need to traverse the entire `nrsack_list`. The sender just needs to traverse the send buffer to free corresponding memory space reported by this block. Similarly, in `pel_nrsacks_merge()`, if $\mathrm{S_{new}} > \mathrm{E_{end}}$, the sender just needs to add the NR-SACK block $(\mathrm{S_{new}} - \mathrm{E_{new}})$ to the tail of the `nrsack_list`. Also, if $\mathrm{S_{new}} = \mathrm{E_{end}}$, the sender just needs to update $\mathrm{E_{end}} = \mathrm{E_{new}}$. Assuming the possibility of shortcut hit is p, then the worst case running time of `pel_nrsacks_can_free()` and `pel_nrsacks_merge()` are decreased to $\mathrm{O(m + (1 - p) * n)}$ and $\mathrm{O((1 - p) * n)}$, respectively.

By adding this `nrsack_list` structure and above processing functions, the mismatch between the information in the `sk_buffs` and the actual data is managed. More importantly, depending on the state information in *sacked* of each skb, a TCP sender always has the correct estimate of the network state.

We extended the Linux kernel (version 3.2.60) to process TCP NR-SACKs at the data sender. We performed an experiment and thanks to promising results, we will be performing a second experiment. Experiment I was in our lab with a simple test-bed.

Based on the positive results of experiment I, we then received authorization to perform an experiment to evaluate TCP NR-SACKs over real satellite link at CNES (Centre National d'Études Spatiales, French government space agency). These experiments are discussed in sections 2.5 and 2.6, respectively.

2.5 Experimental Design I

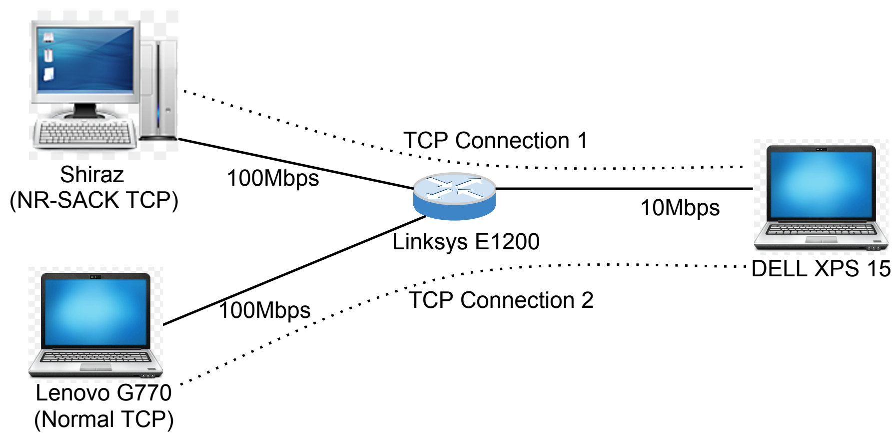

Figure 2.17: Test-bed Topology I

The test-bed (Figure 2.17) of experiment I is composed of a Cisco Linksys E1200 router and three computers. Shiraz and Lenovo G770 are TCP senders, and DELL XPS 15 is the TCP receiver. Shiraz supports TCP NR-SACKs, and Lenovo G770 runs normal TCP. Both TCP senders are connected to the router with a tethered 100Mbps Ethernet cable, and the TCP receiver is connected to the router with a tethered 10Mbps Ethernet cable. Two TCP connections can be established: one between Shiraz and DELL XPS 15, and the other between Lenovo G770 and DELL XPS 15. The traffic is generated by transferring a 50MB file from TCP senders to the receiver over these connections. At any given time, only one TCP connection is transferring the data.

2.5.1 Experimental Parameters

The default upper limit of the TCP send buffer size on Shiraz/LenovoG770 is 905KB (specified by `sysctl_tcp_wmem[2]`). The performance of NR-SACKs are tested under six different send buffer sizes {22KB, 44KB, 90KB, 181KB, 362KB, 905KB}, three different loss rates {0%, 1%, 5%} and three different delays {10ms, 50ms, 500ms}. The extra loss and delay are configured on the outgoing direction of the senders' Ethernet interfaces by using the Linux traffic control [51].

2.5.2 Results

To evaluate the performance of TCP data transfers with NR-SACKs vs. without NR-SACKs, we employ the metric *throughput gain* defined in [19] as $(T_{NR-SACK} - T)/T * 100\%$ where $T_{NR-SACK}$ is the throughput achieved with NR-SACKs and T is the throughput achieved without NR-SACKs for an identical set of experimental parameters (send buffer size, loss rate, bandwidth, and delay). Throughput gain represents the percentage of improvement that results from using NR-SACKs. We also use a *region of gain* [19] defined as the send buffer size interval, [a, b], where any send buffer size between a and b results in an expected throughput gain of at least 5%.

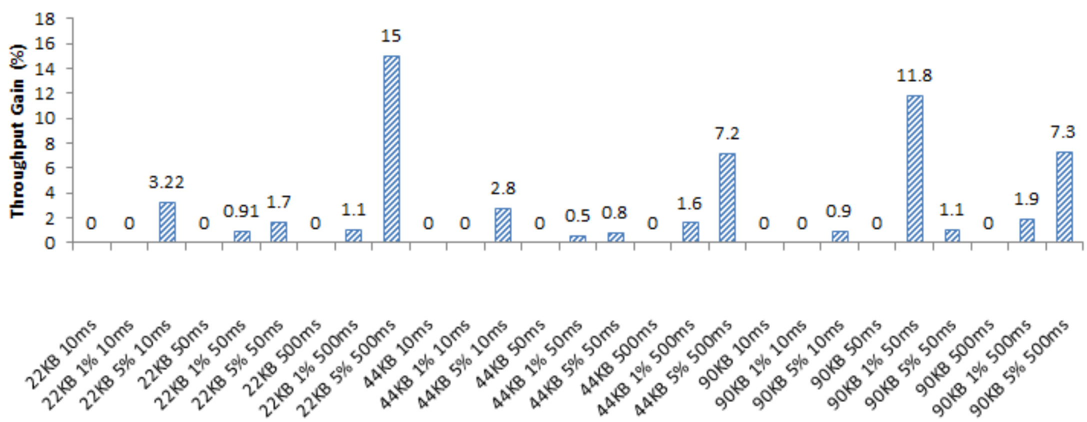

Figure 2.18: Throughput Gain with NR-SACKs (22KB, 44K, 90KB send buffer sizes)

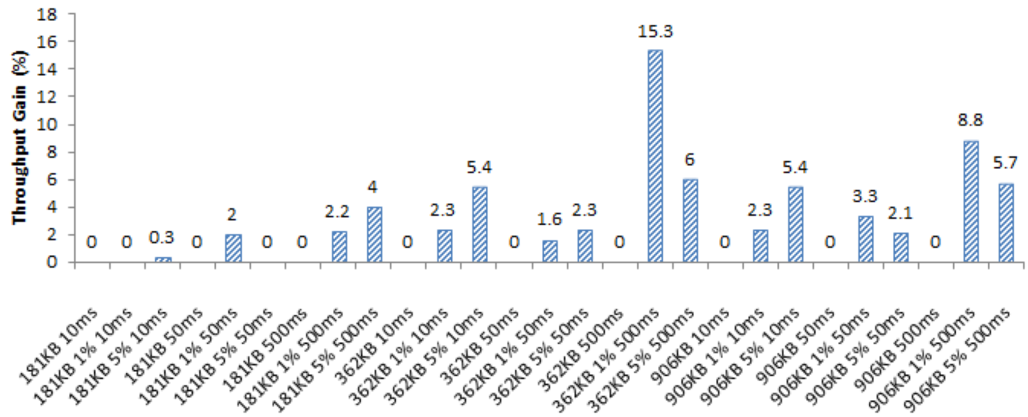

Figure 2.19: Throughput Gain with NR-SACKs (181KB, 362KB, 905KB send buffer sizes)

NR-SACKs require extra processing time at a TCP sender. Our hypothesis was that this overhead would be negligible, that TCP data transfers with NR-SACKs would always perform at least as well as those without NR-SACKs [15] and under certain parameter configurations, NR-SACKs would improve the end-to-end throughput. Figures 2.18 and 2.19 show the throughput gains for all parameter combinations tested. With no loss, the number of runs was one or two because results were identical. With loss being random, the number of runs was at least 30. We observed when no loss was introduced, no NR-SACKs were generated and throughput gain was always 0. We also observed throughput gains were zero or positive for all parameter combinations tested. Our hypothesis was confirmed.

As stated in section 2.2, NR-SACKs can improve the end-to-end throughput when send buffer blocking occurs (i.e., the send buffer is filled by Retransmission Queue (RtxQ)). A RtxQ comprises PDUs which have been sent but not arrived at the receiver, and these PDUs can be either "in flight" or lost. The size of the RtxQ is bounded by both the Bandwidth-Delay Product (BDP) and the average cwnd (denoted $\overline{\text{cwnd}}$):

$$\text{RtxQ size} \leq \min\left(\text{BDP}, \overline{\text{cwnd}}\right) \tag{2.1}$$

For a given delay, increased loss results in a smaller $\overline{\text{cwnd}}$. For a given loss rate, longer delay results in a larger BDP. Impacts of loss rate and delay on throughput gain of NR-SACKs are discussed in sections 2.5.3 and 2.5.4, respectively.

2.5.3 Impact of Loss Rate

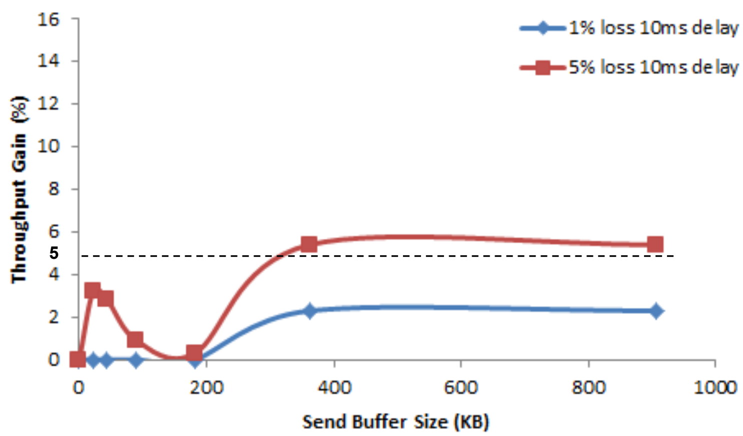

Figure 2.20: Throughput Gain with NR-SACKs (10ms delay)

Figures 2.20, 2.21 and 2.22 show the throughput gains with NR-SACKs when the delay is 10ms, 50ms and 500ms, respectively. From Figure 2.20, we did not observe obvious regions of gain for both loss rates. No send buffer blocking occurred when delay was 10ms for both loss rates. From Figure 2.21, we did not observe region of gain with 5% loss, and region of gain with 1% loss was [65KB, 160KB]. As stated in section 2.5.2, $\overline{\text{cwnd}}$ with 5% loss is smaller than that with 1% loss, and no send buffer blocking occurred with 5% loss. From Figure 2.22, we observed regions of gain for both

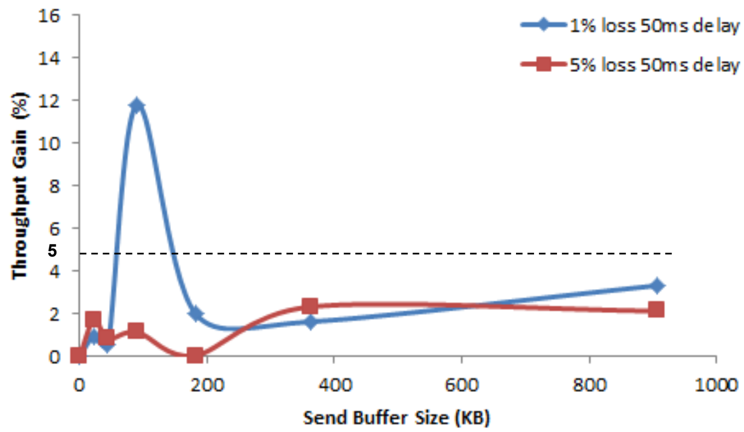

Figure 2.21: Throughput Gain with NR-SACKs (50ms delay)

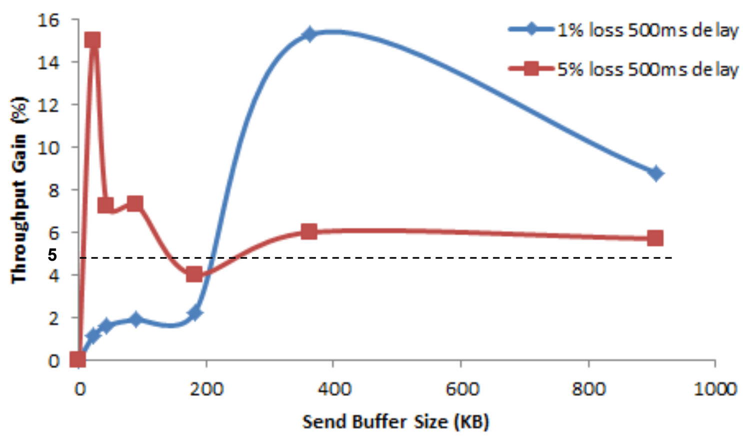

Figure 2.22: Throughput Gain with NR-SACKs (500ms delay)

loss rates. Region of gain for 1% loss was [212KB, 905KB], and that for 5% loss was [10KB, 155KB]. As the loss rate increases, $\overline{\text{cwnd}}$ decreases and hence the send buffer blocking region becomes smaller.

2.5.4 Impact of Delay

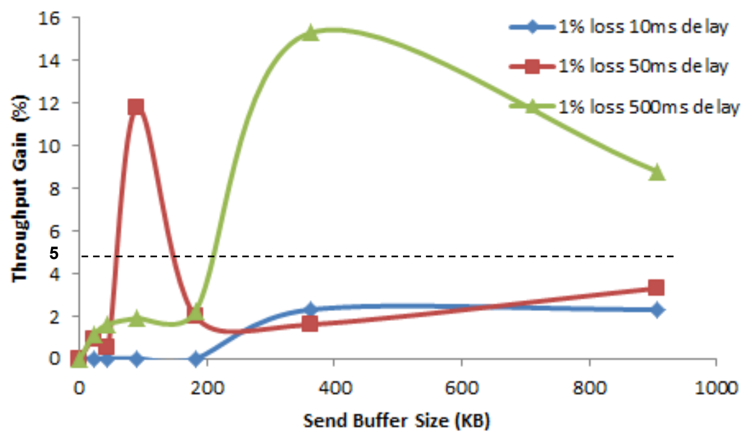

Figure 2.23: Throughput Gain with NR-SACKs (1% loss)

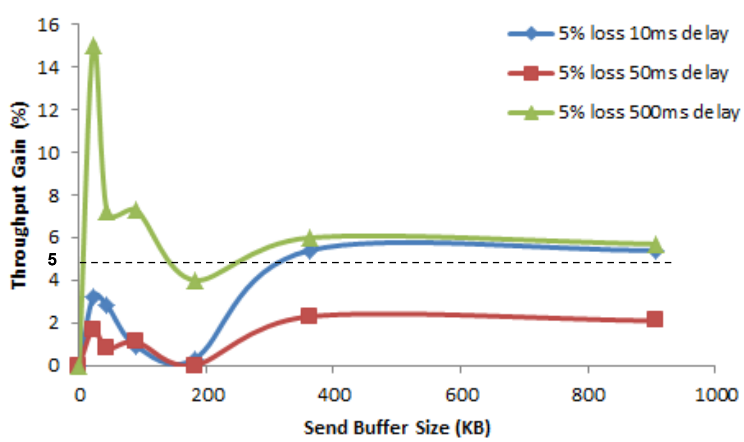

Figure 2.24: Throughput Gain with NR-SACKs (5% loss)

Figures 2.23 and 2.24 show the throughput gain with NR-SACKs when loss rate is 1% and 5%, respectively. From Figure 2.23, we did not observe obvious region of gain with 10ms delay, and regions of gain with 50ms and 500ms delays were [65KB, 160KB] and [212KB, 905KB], respectively. As stated in section 2.5.2, longer delay results in a larger BDP. As delay increases, BDP increases and hence the send buffer blocking region becomes larger. From Figure 2.24, we did not observe obvious regions of gain with 10ms and 50ms delays, and we only observed region of gain with 500ms was [10KB, 155KB].

2.6 Future Work: Experiment Design II

Based on the positive results in our lab, a collaboration study [11] between UD (University of Delaware) and ISAE-SUPAERO (Institut Supérieur de l'Aéronautique et de l'Espace) of quantifying potential gains of TCP NR-SACKs in real long delay, lossy satellite link in CNES is in progress.

ISAE is the French aerospace engineering school in Toulouse, France. SUPAERO is a graduate program within ISAE. SUPAERO covers all the basic engineering disciplines while remaining based on aeronautics and space, the privileged field of application for the most advanced methodologies and techniques. More information about ISAE-SUPAERO can be found at `http://supaero.isae.fr/en/program/supaero_graduate_program`.

Founded in 1961, the Centre National d'Études Spatiales (CNES) is the government agency responsible for shaping and implementing France's space policy in Europe. Its task is to invent the space systems of the future, bring space technologies to maturity and guarantee France's independent access to space. CNES is equivalent to US NASA (National Aeronautics and Space Administration). Figures 2.25 and 2.26 show the satellite control center and terminals in CNES, respectively. More information about CNES can be found at `http://www.cnes.fr`.

Figure 2.27 shows the test-bed. Three computers (one normal TCP sender, one NR-SACK sender and one TCP receiver) are physically located in CNES. The TCP senders and receiver are connected by a real satellite link. The traffic is generated by

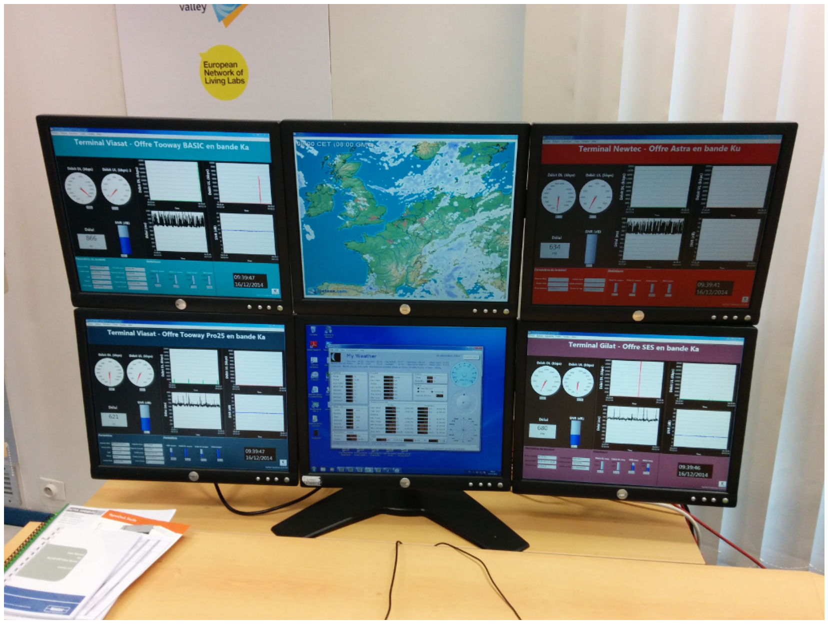

Figure 2.25: Satellite Control Center in CNES

Figure 2.26: Satellite Terminals in CNES

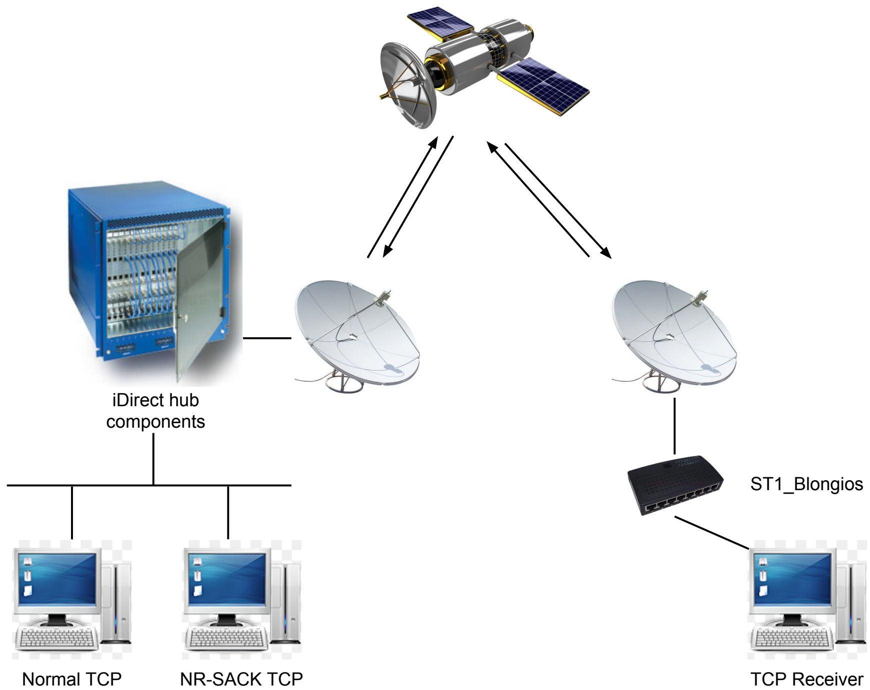

Figure 2.27: Satellite Topology for TCP NR-SACKs in CNES

transferring a 50MB file from a TCP sender to the receiver. The performance of NR-SACKs are proposed to be tested under six different send buffer sizes {22KB, 44KB, 90KB, 181KB, 362KB, 905KB} and four different loss rates {0%, 0.5%, 1%, 5%}.

As part of future work, there is a possible improvement of our implementation. The sender can store `nrsack_blocks` in an array rather than a linked list. Since `nrsack_blocks` are in-order, the sender then can use a binary search to find where to insert/free `nrsack_blocks`. As a result, the worst case running time of `pel_nrsacks_can_free()` can further decrease to $O(m + \log n)$, and that of `pel_nrsacks_merge()` and `pel_nrsacks_clean_nrsacks_by_cumack()` can further decrease to $O(\log n)$.

Chapter 3

NON-RENEGABLE SELECTIVE ACKNOWLEDGMENTS (NR-SACKS) FOR MPTCP

Unlike TCP, the receiver of an MPTCP connection has two level out-of-order queues. Each subflow has its own TCP out-of-order queue to accommodate the received out-of-order TCP-PDUs. At the MPTCP level, an out-of-order queue is also needed to hold the received out-of-order MPTCP-PDUs. In this chapter, we introduce Non-Renegable Selective Acknowledgments (NR-SACKs) to MPTCP and investigate their impact[1] in situations where an MPTCP receiver never discards received out-of-order MPTCP-PDUs (i.e., an MPTCP receiver that, in this chapter, never renegs). Note that, we only enable NR-SACKs at the MPTCP level. Changing TCP to include NR-SACKs is investigated in chapter 2. Investigating the impact of enabling NR-SACKs both at the MPTCP and the TCP subflow levels is beyond the scope of this dissertation, and described in our future work.

3.1 GapAck-Induced Send Buffer Blocking in MPTCP Unordered Data Transfer

Consider a scenario where an MPTCP receiver never renegs. In Figure 3.1, two subflows have been established. After some initial period of data transfer (not shown), assume both subflows have reached their congestion avoidance phase, and the two subflows have roughly the same RTT and the same MSS of 1400 bytes. The MPTCP send buffer, denoted by the blue rectangular box, is assumed to hold up to 11200 bytes of application data. The entire send buffer is equally divided into 8 pieces (each 1400

[1] Results reported in this chapter is published in [22].

bytes) and each piece is denoted by its starting Data Sequence Number (DSN) inside a small rectangular box.

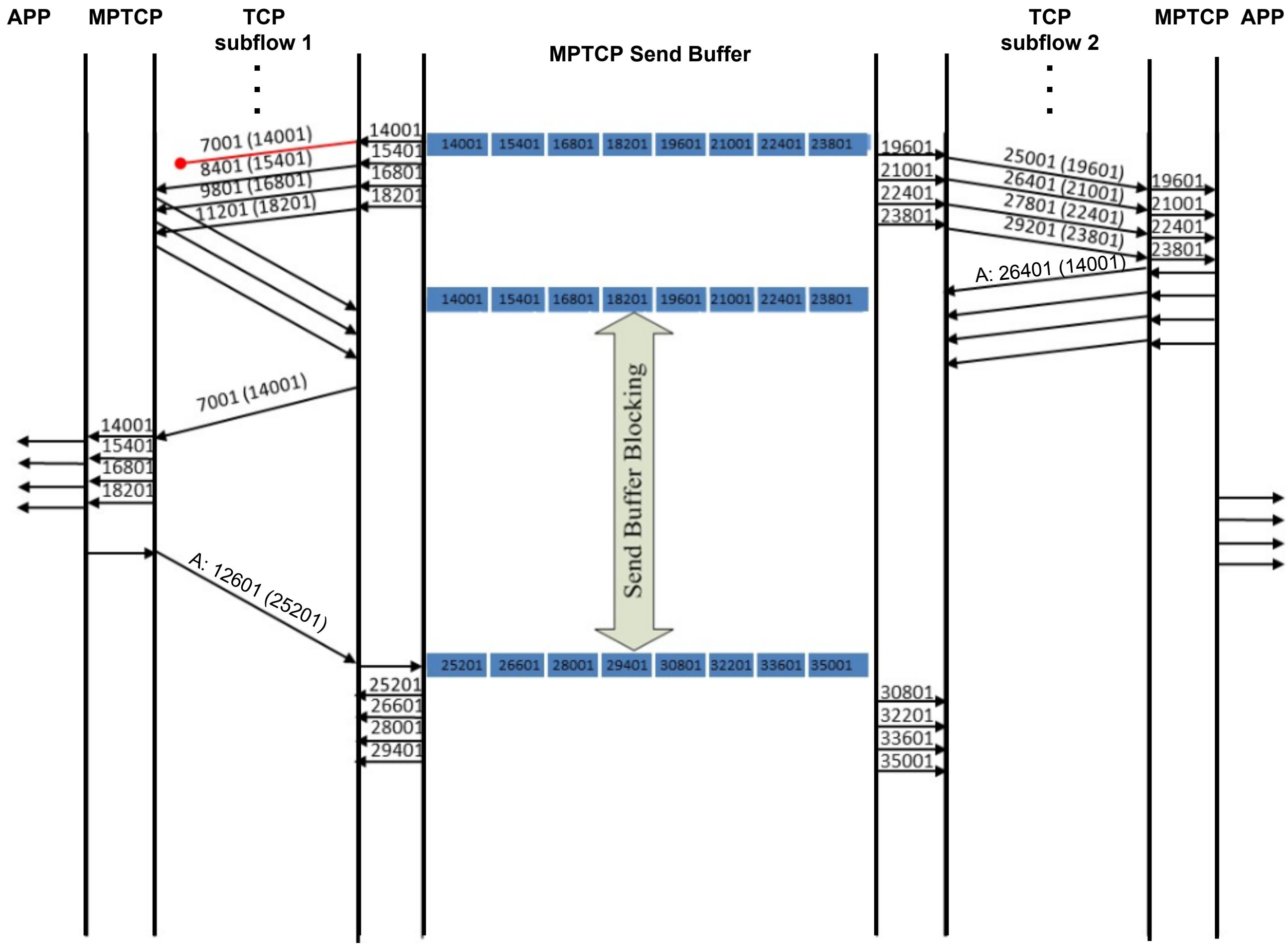

Figure 3.1: Timeline of an Unordered MPTCP Data Transfer

The timeline slice shown in Figure 3.1 starts at a point in the data transfer when both subflows have cwnd = 4. When bytes are then to be transmitted on a subflow, the bytes are encapsulated into TCP-PDUs which are denoted by both the respective subflow's TCP sequence number (S) and the DSN (inside the parentheses) of the first byte of the payload. TCP-PDU S: 7001 (DSN: 14001) of subflow 1 is assumed lost.

Upon reception of the first ack (A: 26401 (DA: 14001)) on subflow 2, the MPTCP sender could in theory continue to transmit new data on subflow 2, since subflow 2 has available cwnd (i.e. $cwnd - num_{packet\_in\_flight} > 0$). However, the MPTCP send buffer does not have any new data. Actually, before the ack of the retransmission of TCP-PDU S: 7001 (DSN: 14001) arrives at the MPTCP sender, even though data corresponding to DSNs 19601 - 25200 have been successfully received by the MPTCP receiver, the MPTCP sender cannot free these data from the send buffer since the DATA ACK does not advance. This scenario illustrates *GapAck-Induced send buffer blocking* (hereafter called send buffer blocking). Send buffer blocking occurs when the total cwnds of all subflows are greater than the MPTCP send buffer size. Send buffer blocking prevents the MPTCP sender from fully utilizing the cwnds of subflows.

In the case where an MPTCP receiver never renegs, this simple timeline illustrates the following:

- After bytes have been received out-of-order by an MPTCP receiver, maintaining these data in the MPTCP send buffer is *unnecessary*, i.e., a waste of memory.
- When send buffer blocking occurs, the MPTCP send buffer size becomes a throughput bottleneck.

3.2 MPTCP Unordered Data Transfer with NR-SACKs

We propose using NR-SACKs to enable an MPTCP receiver to inform an MPTCP sender about the reception and 'non-renegability' of out-of-order data. The details of the proposed modified DSS option which supports NR-SACKs can be found in Appendix A.

Figure 3.2 is analogous to Figure 3.1's example, this time using NR-SACKs. The MPTCP sender and receiver are assumed to have previously negotiated using NR-SACKs during the connection establishment. As in Figure 3.1, TCP-PDU S: 7001(DSN: 14001) of subflow 1 is presumed lost. Notice the difference that the first three acks on subflow 1 and the first four acks on subflow 2 will carry NR-SACK information. When the first ack on subflow 2 arrives, the MPTCP sender is informed

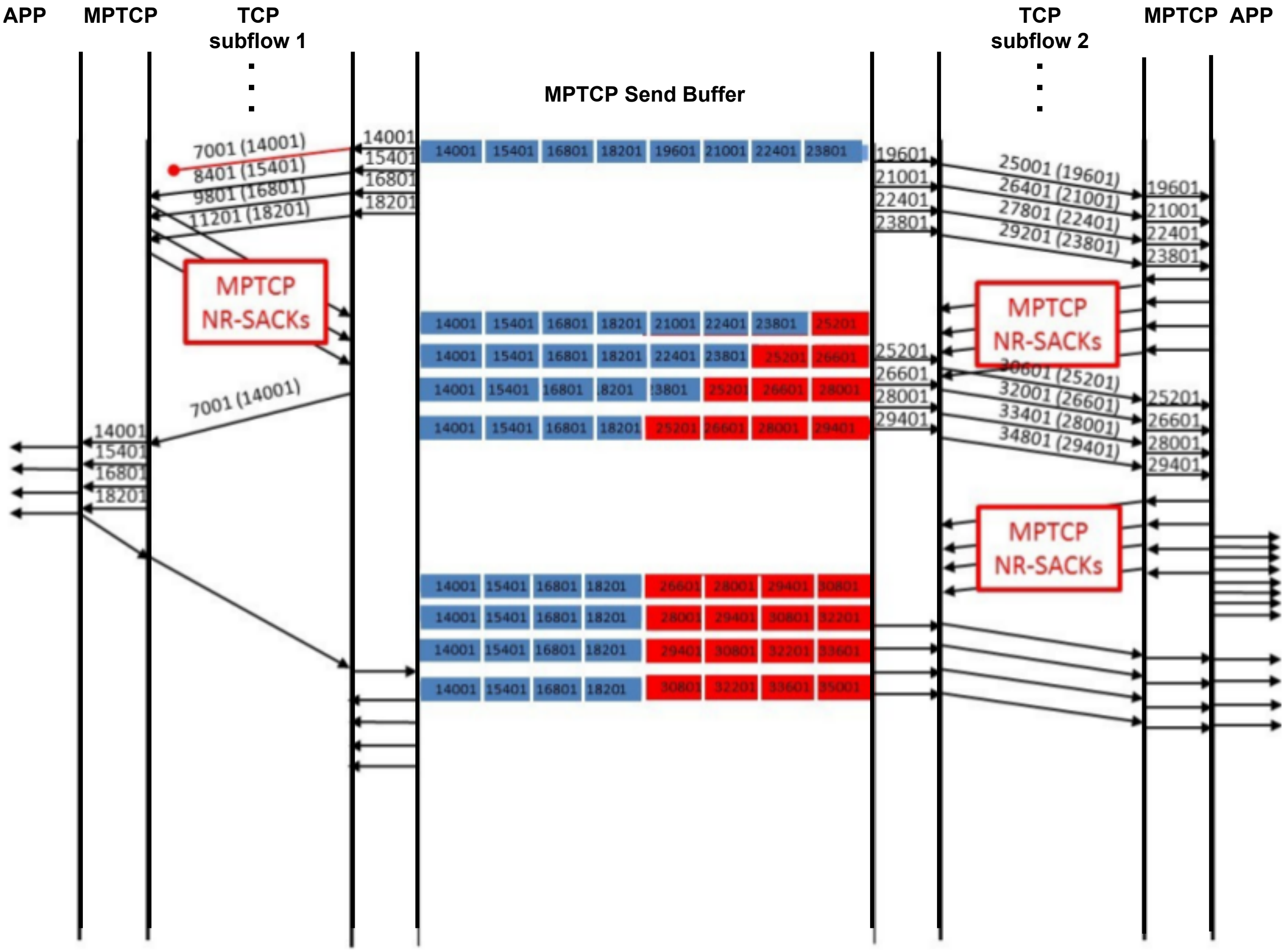

Figure 3.2: Timeline of an Unordered MPTCP Data Transfer with NR-SACKs

that data corresponding to DSNs from 19601 to 21000 have been received and are non-renegable. Unlike Figure 3.1 where the MPTCP sender must retain 19601 - 21000 in case the receiver renegs, here in Figure 3.2 the MPTCP sender immediately frees these NR-SACKed data from the MPTCP send buffer, allowing the application to write new data to the MPTCP send buffer. This new data is transmitted on subflow 2 which has available cwnd. Then the first ack on subflow 1 arrives, but NR-SACK information in this ack is same as the first ack of subflow 2. On the reception of the second, third and fourth acks on subflow 2, more new data are sent out.

Figure 3.2 illustrates the following observations on MPTCP data transfers with NR-SACKs (i.e., where the receiver guarantees not to reneg on the received out-of-order data):

- The MPTCP send buffer only contains *necessary* data (i.e., those data which have not been received by the receiver), thus, NR-SACKs allow a more efficient MPTCP send buffer usage.
- Although subflow 1 is blocked due to the loss, new application data can still be transmitted on subflow 2. NR-SACKs alleviate send buffer blocking hence higher throughput is achieved in Figure 3.2's scenario than Figure 3.1' scenario.

3.3 Implementation

Implementing MPTCP NR-SACKs in the Linux kernel is challenging. We need to thoroughly understand the TCP implementation (roughly 100K lines of code written in C in Linux 3.3) before we can start. A good thing is TCP has a procedure of generating SACKs when out-of-order TCP-PDU are received, and we can imitate this procedure to implement MPTCP NR-SACKs. However, implementation is only the first step, and debugging our implementation is even more difficult. Even a minor bug (e.g., try to dereference a null pointer) would halt the machine, and we need to boot from the correct kernel, find out the problem and recompile the kernel (the recompiling costs about 10 minutes on our machine). It takes about half a year for us to read the whole TCP implementation and one month to implement the MPTCP NR-SACKs, and debugging costs us three months!

In this section, we introduce the procedure of generating NR-SACKs at the MPTCP receiver and processing NR-SACKs at the MPTCP sender.

3.3.1 Supporting NR-SACKs at the MPTCP Receiver

Figure 3.3 shows the procedure of supporting NR-SACKs at the MPTCP receiver:

tcp_data_queue(): This function is the entrance to process a received TCP-PDU (represented by an `skb` in the Linux kernel). If a received TCP-PDU is in-order at

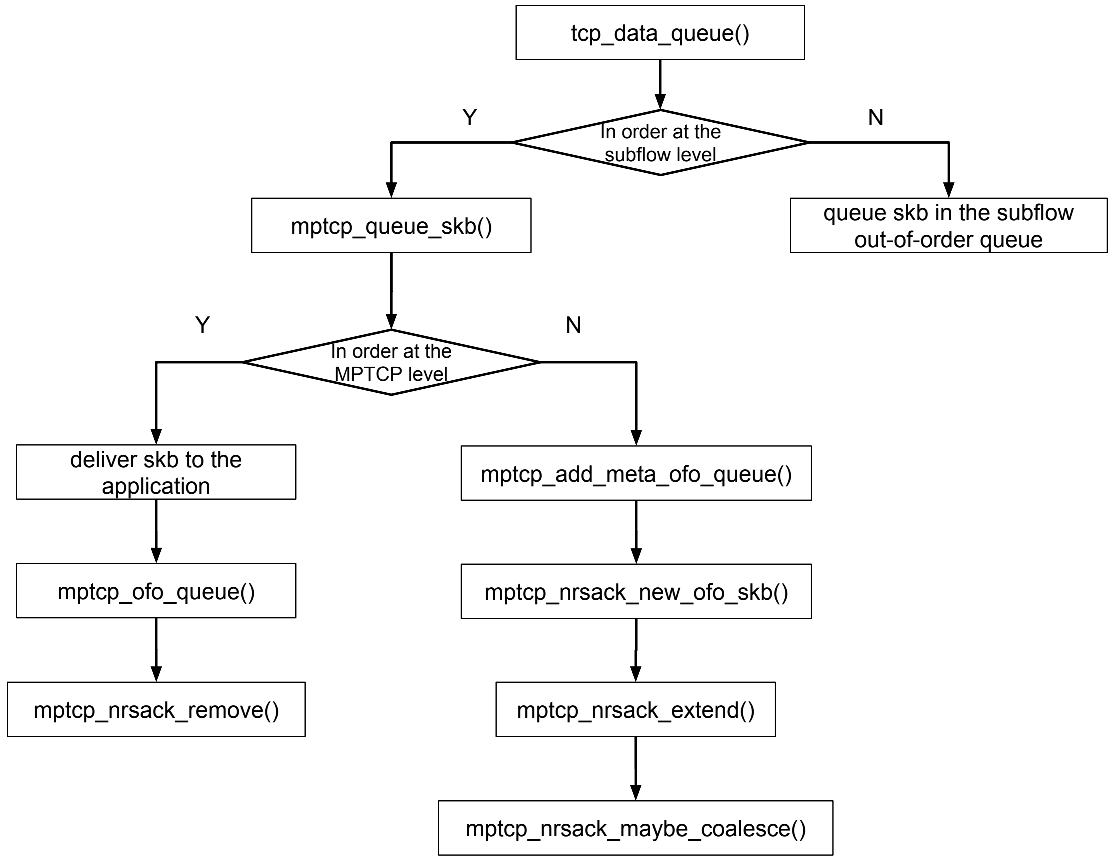

Figure 3.3: Procedure of Supporting NR-SACKs at the MPTCP Receiver

the subflow level, the TCP-PDU is delivered to the MPTCP receiver immediately. Otherwise, the TCP-PDU is queued in the subflow level TCP out-of-order queue.

mptcp_queue_skb(): This function is the entrance to process a received MPTCP-PDU. If a received MPTCP-PDU is in-order at the MPTCP level, the MPTCP-PDU is delivered to the application layer (if possible) or queued in the MPTCP in-order queue. Otherwise, the MPTCP-PDU is queued in the MPTCP level out-of-order queue.

mptcp_ofo_queue(): This function checks whether the received in-order MPTCP-PDU fills the gaps in the MPTCP out-of-order queue. If some gaps are filled, all in-order MPTCP-PDUs are moved from the out-of-order queue to the in-order queue.

For example, Figure 3.4 shows an MPTCP out-of-order queue. At the shown time, MPTCP-PDUs 2000 - 3999, 5000 - 6999 and 8000 - 8999 have been received out-of-order (in shaded boxes). Assume MPTCP-PDU 1000 - 1999 is received, then MPTCP-PDU 2000 - 3999 can be removed from the out-of-order queue and delivered to the application layer (Figure 3.5).

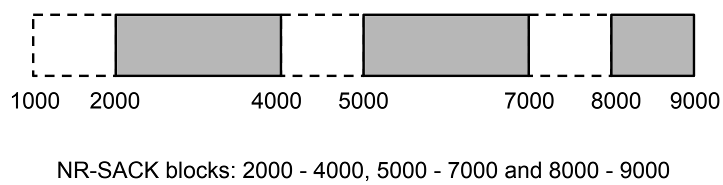

Figure 3.4: An Example MPTCP Out-of-order Queue

mptcp_nrsack_remove(): This function updates NR-SACK blocks correspondingly if some MPTCP-PDUs are removed from the MPTCP out-of-order queue by *mptcp_ofo_queue()*. At the shown time in Figure 3.4, the NR-SACKs are 2000 - 4000, 5000 - 7000 and 8000 - 9000. After MPTCP-PDU 1000 - 1999 is received and MPTCP-PDU 2000 - 3999 is moved from the out-of-order queue, the NR-SACKs are updated to 5000 - 7000 and 8000 - 9000 (Figure 3.5).

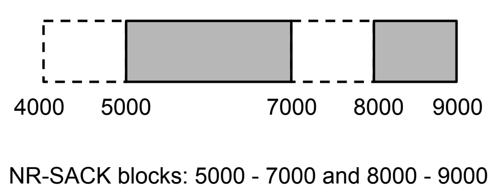

Figure 3.5: MPTCP Out-of-order Queue after MPTCP-PDU 1000 - 1999 is received

mptcp_add_meta_ofo_queue(): This function is the entrance to process a received out-of-order MPTCP-PDU.

mptcp_nrsack_new_ofo_skb(): This function generates an NR-SACK block for a newly received out-of-order MPTCP-PDU. At the shown time in Figure 3.5, assume MPTCP-PDU 7000 - 7999 is received, NR-SACK block 7000 - 8000 is generated correspondingly (Figure 3.6).

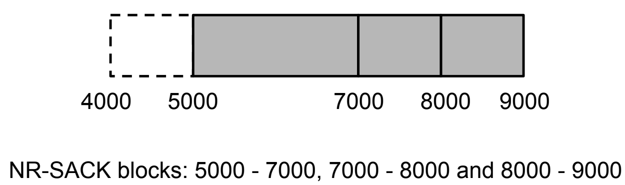

Figure 3.6: MPTCP Out-of-order Queue after MPTCP-PDU 7000 - 7999 is received

mptcp_nrsack_extend(): This function checks whether it is possible to extend existing NR-SACK blocks with the newly generated NR-SACK block. At the shown time in Figure 3.6, after NR-SACK block 7000 - 8000 is generated, NR-SACK block 5000 - 7000 can be extended to 5000 - 8000.

mptcp_nrsack_maybe_coalesce(): This function checks whether newly extended NR-SACK blocks can be merged with the existing NR-SACK blocks. At the shown time in Figure 3.6, after NR-SACK block 5000 - 7000 is extended to 5000 - 8000, it can be merged with NR-SACK block 8000 - 9000. Finally, there is only one NR-SACK block 5000 - 9000.

Whenever an ack is sent out on the subflow level, the latest NR-SACK blocks included in the TCP option field of the ack.

3.3.2 Supporting NR-SACKs at the MPTCP Sender

Supporting NR-SACKs at the MPTCP sender is simple, and the procedure is shown in Figure 3.7:

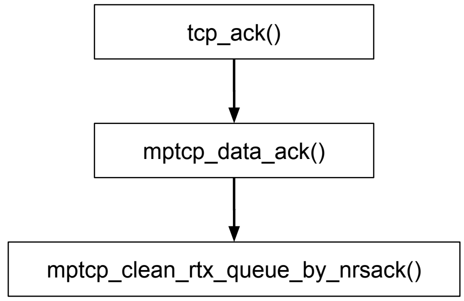

Figure 3.7: Procedure of Supporting NR-SACKs at the MPTCP Sender

tcp_ack(): This function is the entrance to process a received ack at the subflow level. If an ack contains MPTCP information (e.g., DATA ACK, NR-SACKs, etc.), these information are delivered to the MPTCP level.

mptcp_data_ack(): This function processes MPTCP information contained in the received acks. For example, an MPTCP sender frees the send buffer with the DATA ACK.

mptcp_clean_rtx_queue_by_nrsack(): This function frees the send buffer with received NR-SACKs.

3.4 Experimental Setup

We extended the Linux kernel MPTCP implementation [35] to transmit and process NR-SACKs at the data receiver and data sender, respectively. The experiment evaluates the performance of MPTCP data transfers (with two subflows) with NR-SACKs vs. without NR-SACKs under various conditions (path loss rate, delay and send buffer size). The coupled congestion control option [3] is disabled in this evaluation since we want to focus on the impact of NR-SACKs.

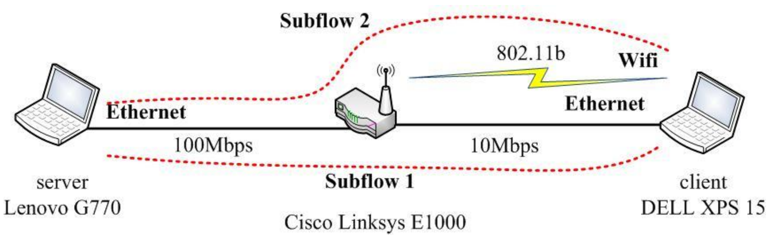

Figure 3.8: Test-bed Topology

3.4.1 Test-bed Topology

The test-bed (Figure 3.8) is composed of a Cisco Linksys E1000 router and two laptops running Ubuntu 11.10. A server is connected to the router with a tethered 100Mbps Ethernet cable. A multihomed client is connected to the router by both an Ethernet cable and a wireless link. To prevent the link between the server and the router being a bottleneck, the Ethernet cable connecting the client and the router has a 10Mbps capacity, and 802.11b (maximum raw data rate is 11Mbps) is used for the wireless link. An MPTCP connection with two subflows is created. Subflow 1 is a TCP connection established over the wired-wired path, and subflow 2 is a TCP connection established over the wired-wireless path. The traffic is generated by moving a 1.46GB file from server to client with MPTCP.

3.4.2 Experimental Parameters

In our experiments, four different delays {5ms, 10ms, 50ms, 500ms} and three different loss rates {0.5%, 1%, 5%} are configured on the outgoing direction of the server's Ethernet interface by using the Linux traffic control [51]. The performance of NR-SACKs has been tested for Linux MPTCP send buffers ranging in size from 14KB to 899KB.

3.5 Results

To evaluate the performance of MPTCP data transfers with NR-SACKs vs. without NR-SACKs, we employ the metric *throughput gain* defined in [19] as $(T_{NR-SACK} -$

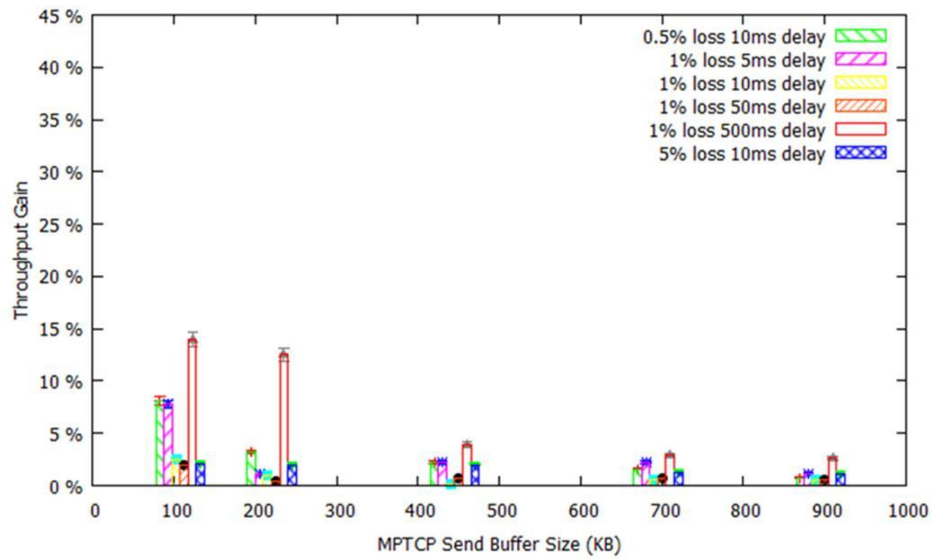

Figure 3.9: Throughput Gain with NR-SACKs (899KB, 700K, 449KB, 224KB, 112KB send buffer sizes)

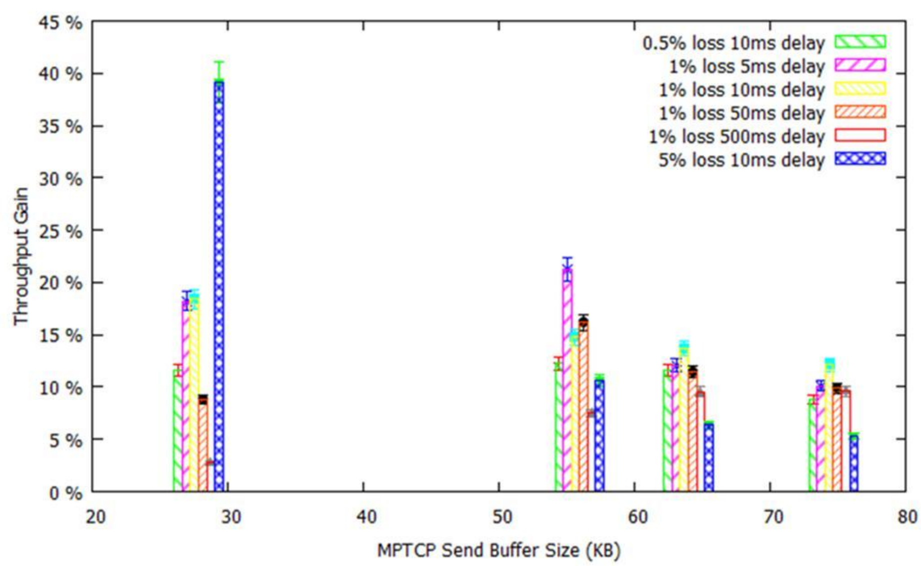

Figure 3.10: Throughput Gain with NR-SACKs (74KB, 64KB, 56KB, 28KB send buffer sizes)

$T)/T * 100\%$ where $T_{NR-SACK}$ is the throughput achieved with NR-SACKs and T is the throughput achieved without NR-SACKs for an identical set of experimental parameters (send buffer size, loss rate, bandwidth, and delay). Throughput gain represents the percentage of improvement that results from using NR-SACKs. We also use a *region of gain* [19] defined as the send buffer size interval, [a, b], where any send buffer size between a and b results in an expected throughput gain of at least 5%.

NR-SACKs require a minimal amount of additional processing time at both end hosts, and a few (roughly 0 - 20 bytes per PDU depending on the number of NR-SACK blocks) extra bytes on the wire. Thus, our first hypothesis was that these overheads would be negligible, and that MPTCP data transfers with NR-SACKs would always perform at least as well as those without NR-SACKs. Figures 3.9 and 3.10 show the throughput gain for a representative subset of the parameter combinations tested. Send buffer sizes in the subset comprises 112KB, 224KB, 449KB, 700KB and 899KB. 899KB is the default MPTCP send buffer size in Ubuntu 11.10. We can see the throughput gains of all send buffer sizes in both figures are positive, so our first hypothesis is confirmed.

Importantly, as the MPTCP send buffer size decreases, we observe a general trend of increasing throughput gain with NR-SACKs in both Figures 3.9 and 3.10. Based on the previous discussion, NR-SACKs can free received out-of-order data from the send buffer prior to (i.e., sooner than) the arrival of the corresponding cum-ack. When send buffer blocking occurs, the total cwnds of all subflows, and hence RtxQ, grow large enough to fill the entire send buffer. NR-SACKs allow more new application data be transmitted. Therefore, our second hypothesis was that when send buffer blocking occurs, MPTCP data transfers with NR-SACKs would outperform those without.

3.5.1 Retransmission queue evolution

To confirm our second hypothesis and gain insight into the send buffer blocking, consider how the Retransmission Queue (RtxQ) size varies over time. Figures 3.11 and 3.12 show how the RtxQ size varies for send buffer sizes 899KB and 28KB, respectively. In both figures, the loss rate and delay on the outgoing direction of the server's interface are 1% and 10ms, respectively. In Figure 3.11, the RtxQ size never reaches 899KB,

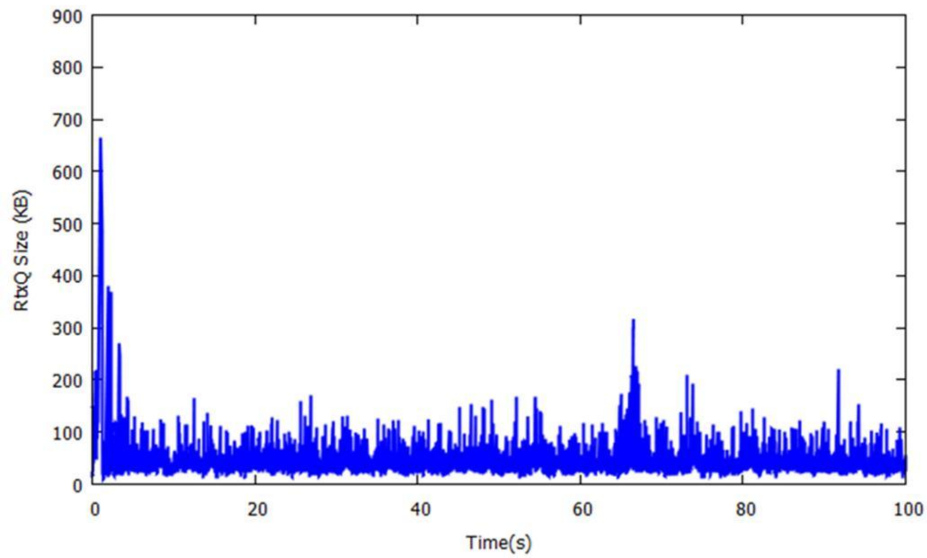

Figure 3.11: Retransmission Queue Evolution without NR-SACKs (899KB send buffer size, 1% loss, 10ms delay)

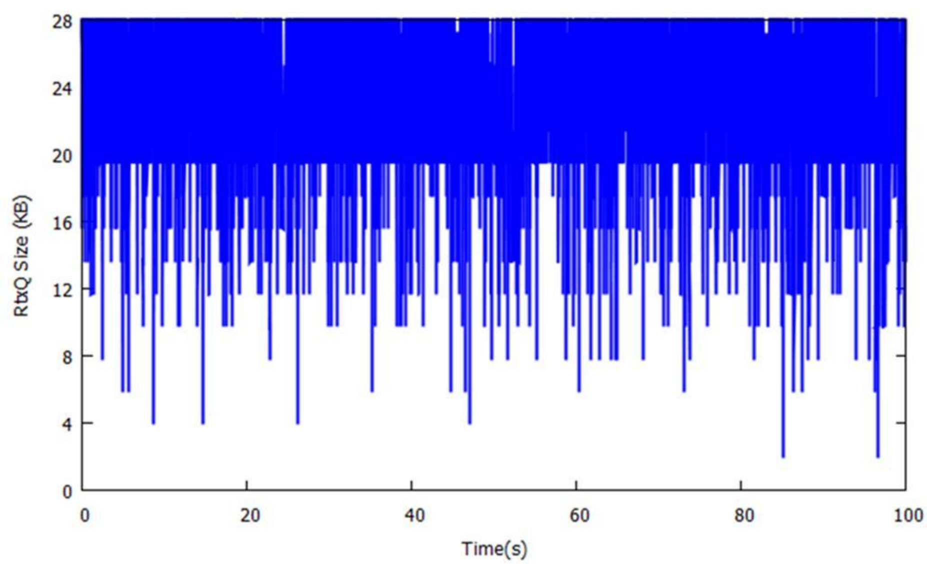

Figure 3.12: Retransmission Queue Evolution without NR-SACKs (28KB send buffer size, 1% loss, 10ms delay)

thus no send buffer blocking occurs, and no throughput gain is expected by using NR-SACKs (as confirmed in Figure 3.9). In Figure 3.12, the RtxQ size frequently reaches 28KB, each time causing send buffer blocking. When send buffer blocking occurs, significant throughput gain is expected by using NR-SACKs (as confirmed in Figure 3.10). These results confirm our second hypothesis.

3.5.2 Impact of Loss Rate

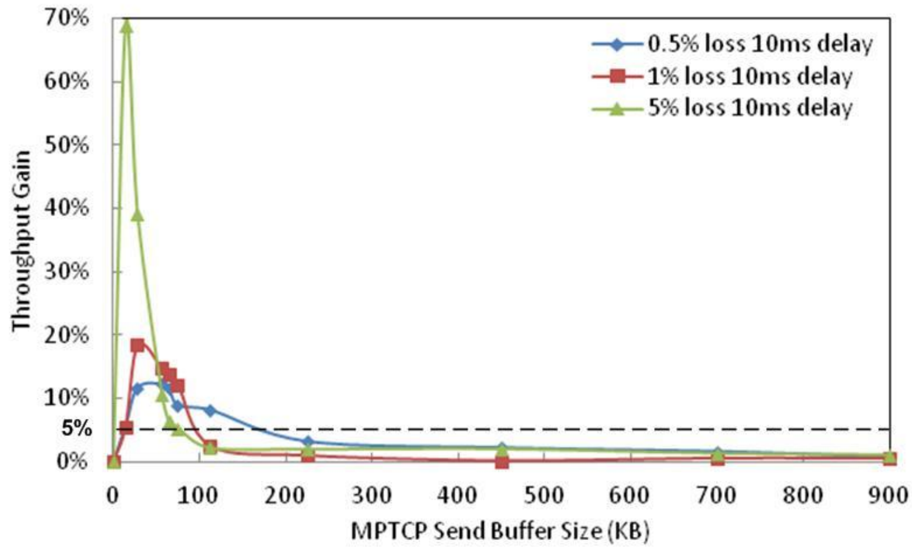

Figure 3.13: Throughput Gain with NR-SACKs (same delay different loss rates)

From Figure 3.13, we observed: as the loss rate increases, (i) the right edge of region of gain moves to the left, and (ii) the maximum throughput gain in the region of gain moves up. The reason for observation (i) is: higher loss rates result in smaller total cwnds (and hence smaller RtxQ size), so right edge of the region of send buffer blocking shrinks. The reason for observation (ii) is: higher loss rates make an MPTCP receiver generate more NR-SACK information (and hence more received out-of-order data can be freed from the send buffer), so the throughput gain increases.

3.5.3 Impact of Delay

For a given bandwidth, longer delays result in a larger Bandwidth-Delay Product (BDP). When the BDP < MPTCP send buffer size, no send buffer blocking occurs since the total cwnd size is bounded by the BDP. Send buffer blocking occurs only when BDP

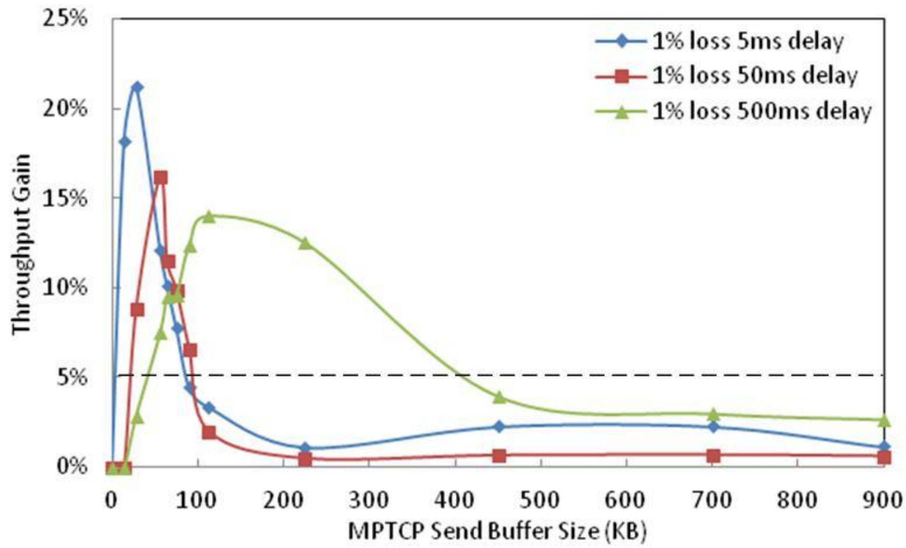

Figure 3.14: Throughput Gain with NR-SACKs (same loss rate different delay)

$\geq$ MPTCP send buffer size. Therefore, we hypothesized that gains from MPTCP with NR-SACKs would be larger for a longer delay than for a shorter delay.

Our hypothesis is confirmed by Figure 3.14. As the delay increases, the right edge of region of gain moves right. For all loss rates tested, we also observed that the throughput gain with NR-SACKs is greater over a link with a shorter delay (consistent with the results in [19]).

3.6 Conclusion

In this chapter, we introduced NR-SACKs to MPTCP and investigated their impact in situations where an MPTCP receiver never renegs. We extended the Linux MPTCP implementation to support NR-SACKs. The experiment setup was extremely limited (only one topology). However, these preliminary results show that (i) MPTCP data transfers with NR-SACKs never perform worse than those without NR-SACKs, and (ii) NR-SACKs can improve end-to-end throughput in MPTCP when send buffer blocking occurs. In an MPTCP connection with several high-BDP subflows, send buffer blocking can occur and seriously decrease the end-to-end throughput. NR-SACKs can alleviate the send buffer blocking and achieve higher throughput. We propose an extensive experiment with more topologies and parameter (e.g., loss rate, delay, etc) combinations setup in our future work. Based on the argument that the design to tolerate reneging is wrong, preliminary results would indicate that NR-SACKs SHOULD be added to the MPTCP standard.

Chapter 4

HOW TO DERIVE A GOOD SCHEDULER FOR MPTCP

In this chapter, we first describe problems with the default scheduler used by the Linux kernel MPTCP implementation. Then we propose the design of a new scheduler. Experimental results[1] show that our proposed scheduler under some circumstances improves the throughput in MPTCP by alleviating the problems caused by the default scheduler.

4.1 Problems

The Linux kernel MPTCP implementation scheduler can be summarized as:

- when multiple subflows have available cwnd to send data, data is transmitted on the subflow with the shortest estimated smoothed round trip time (srtt).

This default scheduler seems reasonable. Srtt reflects the time between when an MPTCP-PDU is sent out and when its corresponding acknowledgment comes back. The default scheduler selects the 'fastest' subflow to send next MPTCP-PDU.

A scheduler works in cooperation with the congestion control mechanism. The aim of the MPTCP congestion control mechanism is to move data away from congested path(s) [4]. However, if multiple subflows have available cwnd, this aim is difficult to achieve without a good scheduler.

Consider a hypothetical scenario of an MPTCP connection with two subflows as shown in Figure 4.1. Subflow 1 is established on a 3G path with a large buffer (can queue up to 200 PDUs), resulting in possible longer RTT delays but lower drop rates. Subflow 2 is established on a Wifi path with a smaller buffer (can queue up to

[1] These results have been published in [32].

20 PDUs), resulting in possible shorter RTT delays but potentially higher drop rates. At the shown time, assume the Wifi path buffer is full, while that of the 3G path is only half full. The Wifi path (subflow 2) is congested, i.e., newly arriving PDUs will be dropped. However, if both subflows have available cwnd, the default scheduler will choose subflow 2 to send the next PDU because subflow 2's srtt is smaller. In this scenario, a subflow's srtt does not reflect the subflow's congestion. As demonstrated by this hypothetical scenario, a scheduler only based on srtt may be **inconsistent with the aim of the MPTCP congestion control mechanism** (problem 1).

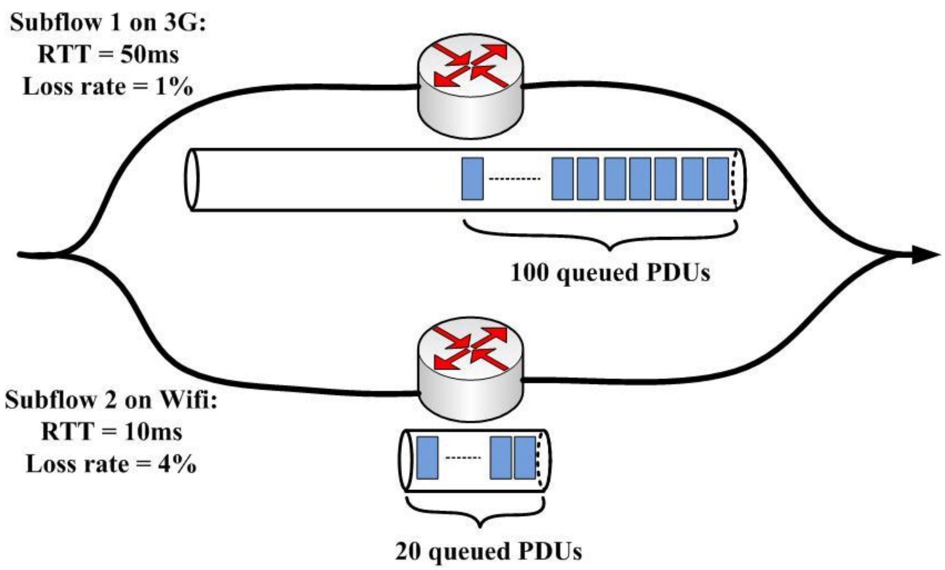

Figure 4.1: A scenario in which RTT and congestion mismatch

In above scenario, sending more PDUs on subflow 2 may cause loss due to congestion. More seriously, once lost PDUs are detected, they need to be retransmitted and subflow 2's cwnd will decrease. It takes time for subflow 2 to increase its cwnd to the value before the loss. This situation causes an **inefficient usage of available path capacity** (problem 2) on subflow 2. In the above scenario, the MPTCP sender could have sent the PDU on subflow 1 rather than subflow 2.

4.2 Analysis

MPTCP employs an additive-increase multiplicative-decrease (AIMD) coupled congestion control mechanism. Each subflow continually increases its cwnd even to a point that exceeds the available path capacity (defined as the maximum number of PDUs in flight of this subflow) before detecting a loss. If the number of outstanding PDUs of a subflow has reached the available path capacity, sending more PDUs through the subflow will cause congestion loss.

To both be consistent with the aim of the congestion control mechanism and using network resources efficiently (solving problems 1 and 2), a good scheduler needs to select a 'fastest' subflow without causing loss because of congestion. We propose, **a scheduler selects a subflow based on not only its srtt but also the subflow's congestion situation.**

A scheduler needs to estimate the congestion situation of each subflow. At any given time, a scheduler knows the number of outstanding MPTCP-PDUs on a subflow. If the available path capacity of each subflow could be accurately estimated, the scheduler could tell whether sending more MPTCP-PDUs on a subflow will reach and/or surpass the subflow's available path capacity. Consider a per-subflow ratio *Occupied* $= (Number\_of\_outstanding\_packets + 1)/Estimated\_path\_capacity$. Define two congestion thresholds, $\gamma < \delta$. For subflow i, when $Occupied_i \leq \gamma$, sending one more MPTCP-PDU on subflow i is considered not to cause congestion on the path. When $\gamma < Occupied_i \leq \delta$, sending one more MPTCP-PDU on subflow i is considered to cause congestion on the path. When $Occupied_i > \delta$, sending one more MPTCP-PDU on subflow i is considered to cause loss because of congestion on the path. Now, the solution seems to be simple. For an MPTCP connection, say with two subflows, if sending one MPTCP-PDU to both subflows will not cause congestion (i.e., $Occupied \leq \gamma$ for both subflows), the scheduler sends the next MPTCP-PDU on the subflow with shorter srtt (i.e., the current default). If sending one MPTCP-PDU to both subflows will cause loss because of congestion (i.e., $Occupied > \delta$ for both subflows), the scheduler delays sending the next MPTCP-PDU temporarily. Otherwise, the less congested subflow is selected.

4.2.1 Techniques

Since the available capacities of network paths are changing all the time, accurately estimating the available path capacity of a subflow is challenging. Actually, the problem of estimating the available path capacity of a TCP connection is not new. The aim of TCP congestion control is to dynamically adapt cwnd size to be roughly the available path capacity. Therefore, we can employ the techniques of TCP congestion control to estimate the available path capacity of a subflow.

Consider what parameters can be used to estimate a subflow's available path capacity. Available parameters include per-subflow cwnd, slow start threshold, and RTT related values (sample RTTs, srtt, RTO values, etc).

An obvious question is why not just use per-subflow cwnd as the estimated available path capacity? As mentioned in the previous subsection, each MPTCP subflow employs a modified AIMD congestion control algorithm. A subflow's cwnd can temporarily exceed the available path capacity before detecting a loss. After detecting a loss, the cwnd decreases and the new slow start threshold is below the available path capacity. Even worse, if the available path capacity is relatively large, several round trips may be needed for the cwnd to reach the available path capacity after a loss. Therefore, using cwnd or slow start threshold as the estimated available path capacity can be imprecise.

What about RTT-related values? In TCP congestion control techniques, one preventive rather than reactive algorithm is end-to-end delay-based congestion avoidance algorithm (DCA) [45]. DCA algorithms keep track of TCP-PDU RTTs (called sample RTTs). An increase in sample RTTs presumes increased queuing delay, thus increased congestion in intermediate routers. TCP-Vegas [43] and FAST [44] employ this technique in their congestion control mechanisms. Can we use changes of sample RTTs to estimate the available path capacity? The answer remains no. As argued in [46], congestion information contained in TCP RTT samples cannot reliably predict packet loss, and thus cannot be used to accurately estimate available path capacity. The reasons are (i) the collected sample RTTs are too coarse to accurately track the bursty congestion associated with packet loss over high-speed paths, and (ii) sometimes short-term queue fluctuations which are not associated with losses make the changes of sample RTTs not reliably reflect the congestion level at the router.

We need a stable method to estimate a subflow's available path capacity. A feasible method is using multiple parameters. Reconsider the AIMD congestion control algorithm used by each subflow. After a loss (assume this loss is caused by congestion) is detected on a subflow, the subflow will decrease its cwnd. The current available path capacity can be expected somewhere between the cwnd size when the loss is detected and the new slow start threshold. Thus the technique of BI-TCP [42] can be used. BI-TCP uses a binary search algorithm where the cwnd grows to the mid-point between the last cwnd size where TCP has a packet loss and the last cwnd size TCP does not have a loss for one RTT.

4.3 A Scheduling Policy Based on Estimated Subflow Path Capacities

Estimating available path capacity of a subflow: Initially, the estimated available path capacity of subflow i is set to a default maximum (a large constant). If a loss is detected on subflow i, the estimated available path capacity is set to the mid-point between the cwnd (i.e., max) before loss detection and the new slow start threshold (i.e., min). After the number of outstanding MPTCP-PDUs of subflow i reaches the estimated available path capacity, if subflow i does not detect further packet loss, it means that the available path capacity is under-estimated. Then a new min is set to the number of outstanding MPTCP-PDUs, and the estimated available path capacity is recalculated. After the number of outstanding MPTCP-PDUs reaches the max, if no loss has been detected, it means that the actual available path capacity has increased since the last loss. Then a new max is set to the current cwnd size, a new min is set to the current number of outstanding MPTCP-PDUs, and a new estimated available path capacity is recalculated. The full proposed algorithm to estimate

if (a loss has been detected since this algorithm was last run and hence *ss_threshold* has been updated) **then**
    $Max = 2 * ss\_threshold$
    $Min = ss\_threshold$ ▷ update both max and min
    $Estimated\_path\_capacity = \frac{1}{2} * (Max + Min)$
else if ($Estimated\_path\_capacity \leq Num\_of\_outstanding < Max$) **then**
    $Min = Num\_of\_outstanding$ ▷ only update min
    $Estimated\_path\_capacity = \frac{1}{2} * (Max + Min)$
else if ($Num\_of\_outstanding \geq Max$) **then**
    $Max = Cwnd$
    $Min = Num\_of\_outstanding$ ▷ update both max and min
    $Estimated\_path\_capacity = \frac{1}{2} * (Max + Min)$
end if

Figure 4.2: Algorithm to Estimate Available Path Capacity of a Subflow

available path capacity is shown in Figure 4.2.

Scheduling policy: When an MPTCP-PDU is ready to be sent, the MPTCP sender estimates the available subflows. $Occupied_i$ is calculated for each subflow i. If the MPTCP-PDU is an MPTCP level retransmission, that PDU will not be re-sent on the subflow used for the original. Otherwise, subflows with $Occupied_i \leq \delta$ can be used to send the MPTCP-PDU. If multiple subflows can be used and some available subflows will not be congested by sending one more MPTCP-PDU (i.e., $Occupied_i \leq \gamma$), the subflow with the shortest srtt is selected. When all available subflows would be congested by sending one more MPTCP-PDU (i.e., $Occupied_i > \gamma$), the subflow with the smallest $Occupied_i$ is selected. The full scheduling algorithm is shown in Figure 4.3.

Just as BI-TCP, our algorithm keeps the available path capacity of a subflow longer at the saturation point. Our proposed scheduler considers both the 'speed' and the congestion situation of each subflow. Thus, this proposed scheduler is consistent with the aim of the congestion control mechanism, and using network resources more efficiently.

```
Each time the scheduler runs:
Min_srtt = 0xFFFFFFFF               ▷ initialize to be the maximal 32-bit unsigned int
Min_occupied = 0xFFFFFFFF           ▷ initialize to be the maximal 32-bit unsigned int
Num_uncongested_path = 0            ▷ initialize number of uncongested subflows
Num_available_path = 0              ▷ initialize number of available subflows

for each subflow i do
    if (next MPTCP-PDU is a retransmission originally transmitted on subflow i) then
        continue
    end if
    if (Num_of_outstanding_i ≥ Cwnd_i) then                  ▷ no available cwnd
        continue
    end if
    Occupied_i = (Num_of_outstanding_i + 1)/Estimated_path_capacity_i
    if (Occupied_i > δ) then                                 ▷ cause congestion loss
        continue
    end if
    Num_available_path++                                     ▷ count as available subflows
    if (Occupied_i ≤ γ) then                                 ▷ uncongested
        Num_uncongested_path++                               ▷ count as uncongested subflows
    end if
end for

if (Num_uncongested_path > 0) then          ▷ select the 'fastest' uncongested subflow
    for each uncongested subflow i do
        if (Srtt_i < Min_srtt) then
            Min_srtt = Srtt_i
            Selected_subflow = i
        end if
    end for
else if (Num_available_path > 0) then       ▷ select the 'least' congested subflow
    for each available subflow i do
        if (Occupied_i < Min_occupied) then
            Min_occupied = Occupied_i
            Selected_subflow = i
        end if
    end for
else
    Selected_subflow = NULL                 ▷ delay the scheduling of next MPTCP-PDU
end if
```

Figure 4.3: Algorithm of Proposed Scheduler

4.4 Implementation

In the Linux MPTCP, the following functions are related to the scheduler:

`mptcp_next_segment`: selects the next MPTCP-PDU to be scheduled.

`get_available_subflow`: selects a subflow to transmit the selected MPTCP-PDU.

`mptcp_write_xmit`: transmits the selected MPTCP-PDU on the selected subflow.

The algorithms in Figures 4.2 and 4.3 are implemented in `get_available_subflow`. At a given time during the transmission, a subflow's sender has following available variables:

`snd_ssthresh`: current slow start threshold;

`snd_cwnd`: current cwnd;

`packets_out`: current number of TCP-PDUs in the retransmission queue (TCP-PDUs which have been sent but not cumulatively acknowledged (cumacked));

`sacked_out`: current number of TCP-PDUs which have been reported as received by Selective Acknowledgments (SACKs) but not yet cumacked;

`fackets_out`: current number of TCP-PDUs which have been forward acknowledged (FACKed) [9];

`retrans_out`: current number of TCP-PDUs which have been retransmitted and the retransmissions are "in flight".

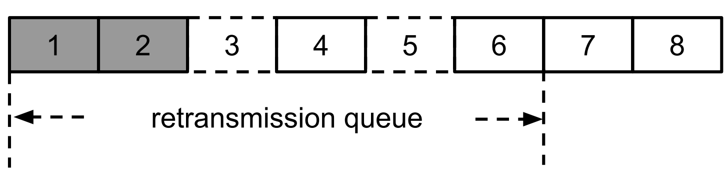

Figure 4.4: A Subflow's Send Buffer During Data Transfer

Let us consider an example to better understand the later four variables. Figure 4.4 shows a subflow's send buffer which contains 8 TCP-PDUs. TCP-PDUs 1 to 6, denoted as retransmission queue, have been sent but not cumacked. TCP-PDUs 3 and 5, denoted by dashed boxes, have been reported as received by SACKs. TCP-PDUs 1 and 2, denoted by shaded boxes, have been retransmitted. TCP-PDUs 7 and 8 are waiting to be transmitted for the first time. At the shown time, `packets_out` $= 6$, `sacked_out` $= 2$, `fackets_out` $= 5$ and `retrans_out` $= 2$.

A scheduler needs every subflow's three variables: *ss_threshold*, *Cwnd* and *Num_of_outstanding*. Obviously, *ss_threshold* and *Cwnd* are already available. Now, the scheduler needs to compute *Num_of_outstanding* from above available information. Can the scheduler directly treat `packets_out` as *Num_of_outstanding*?

Num_of_outstanding is the number of TCP-PDUs which are "in flight" in the network and occupy the available path capacity. However, TCP-PDUs which have not been cumacked include: (i) those which arrive at the receiver out-of-order (*Num_of_ofo*), (ii) those which are lost in the network (*Num_of_lost*), and (iii) those which are "in flight" (include both original transmissions and any retransmissions). `sacked_out` is the best available estimation of *Num_of_ofo*. Note that, `sacked_out` is always $\leq$ *Num_of_ofo*, because some TCP-PDUs can arrive at the receiver out-of-order but the corresponding SACKs are not received by the sender. We use the method in [9] to compute *Num_of_lost* as `fackets_out` $-$ `sacked_out`. For the example in Figure 4.4, the first transmissions of TCP-PDUs 1, 2 and 4 are considered to be lost. Importantly, *Num_of_outstanding* also includes retransmitted TCP-PDUs. Therefore, $\mathit{Num\_of\_outstanding} = \texttt{packets\_out} - \texttt{sacked\_out} - (\texttt{fackets\_out} - \texttt{sacked\_out}) + \texttt{retrans\_out}$ and the final formula is:

$$\mathit{Num\_of\_outstanding} = \texttt{packets\_out} - \texttt{fackets\_out} + \texttt{retrans\_out} \tag{4.1}$$

Now, the scheduler has all necessary variables for the implementation.

Table 4.1: MPTCP Data Transfer without Cross Traffic

	Default	P-(0.5,1.0)	P-(0.7,1.0)	P-(0.9,1.0)	P-(0.9,1.2)	P-(0.9,1.4)
Throughput (MBps)	1.58	1.82	1.81	1.81	1.80	1.59
Retransmissions (in PDUs)	1288	4	18	13	30	838
TCP-PDUs sent on subflow 1	68123	68168	68358	68588	68927	68231
TCP-PDUs sent on subflow 2	45513	44184	44008	43773	43249	44955

4.5 Evaluation Preliminaries

We implemented our proposed scheduler in the Linux kernel, and evaluated the performance of MPTCP data transfers with two subflows with our proposed scheduler vs. with the default scheduler. The comparison was made with and without cross traffic. Obviously, the coupled congestion control option is turned on.

Our test-bed is composed of two Cisco Linksys routers and three laptops running Ubuntu 11.10 (see Figure 4.5). Both laptops 1 and 2 are multihomed by using the tethered Ethernet interface and a Cisco USB Ethernet adapter. An MPTCP connection is established between laptops 1 and 2. Subflow 1 is established between the two tethered Ethernet interfaces, while subflow 2 is established between the two Cisco USB Ethernet adapters. Each Cisco USB Ethernet adapter has a small internal buffer that can queue up to 3 TCP-PDUs, thus the available path capacity of subflow 2 is less than that of subflow 1. A TCP connection is established between laptops 2 and 3. Our test-bed topology, as well as the speed of each link, are shown in Figure 4.5. The MPTCP traffic is generated by moving a 150MB file from laptop 1 to laptop 2, while the cross traffic is generated by moving a file of unbounded size from laptop 3 to laptop 2 with TCP.

4.6 Performance Evaluation

4.6.1 Results without Cross Traffic

Based on the analysis in Section II, we hypothesized our proposed scheduler uses network resources more efficiently than the default scheduler. Table 4.1 shows the

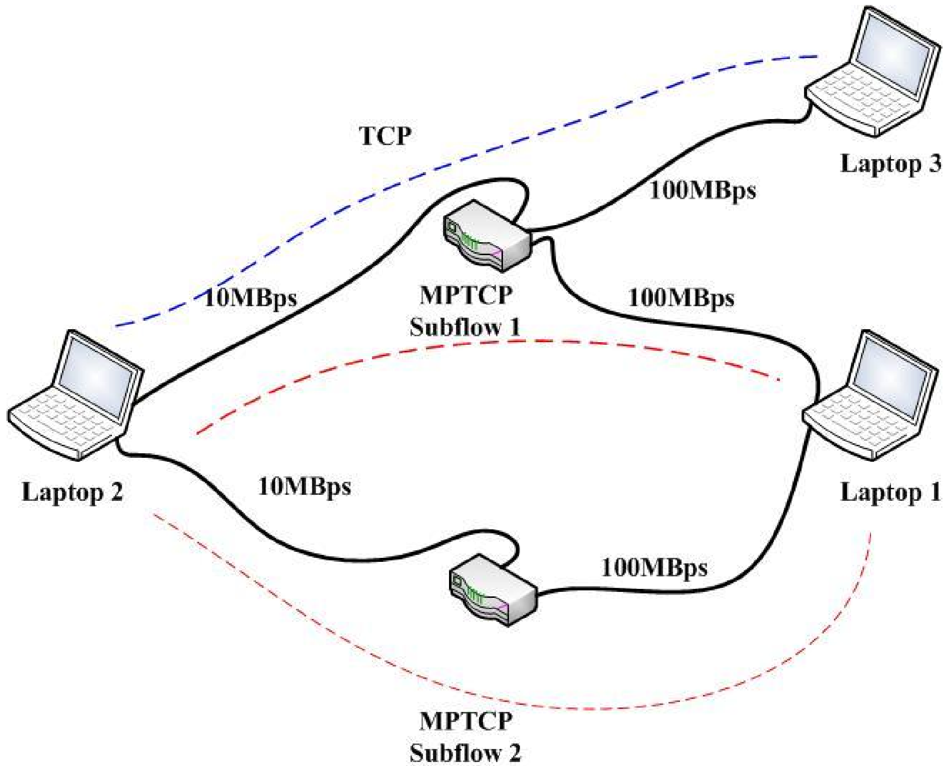

Figure 4.5: Test-bed Topology

results for the 150MB data transfer without cross traffic. Each entry in Table 4.1 represents the average of ten transfers. The different schedulers are the default scheduler (denoted Default) and versions of proposed scheduler with different combinations of γ and δ. For example, a version of proposed scheduler with $\gamma = 0.5$ and $\delta = 1.0$ is denoted P-(0.5,1.0).

Since no artificial loss is introduced, all losses are caused by congestion. The default scheduler causes more retransmissions than our proposed scheduler with $\delta = 1.0$. Figures 4.6 show the one way delay of subflow 1 between 80s and 84s of data transfer with the default scheduler and with scheduler P-(0.9,1.0). The default scheduler sends data to subflow 1, and its one way delay increases (which means the queue length in the router increases) until the the number of outstanding TCP-PDUs exceeds the available path capacity and congestion loss occurs. The sender detects these losses and

Table 4.2: MPTCP Data Transfer with Cross Traffic

	Default	P-(0.5,1.0)	P-(0.9,1.0)	P-(0.5,1.2)	P-(0.9,1.2)
Throughput (MBps)	0.92	0.99	0.98	0.97	0.96
Retransmissions (in PDUs)	1684	11	10	47	29
TCP-PDUs sent on subflow 1	44459	32060	31270	32138	20110
TCP-PDUs sent on subflow 2	69573	80299	81086	80257	92267

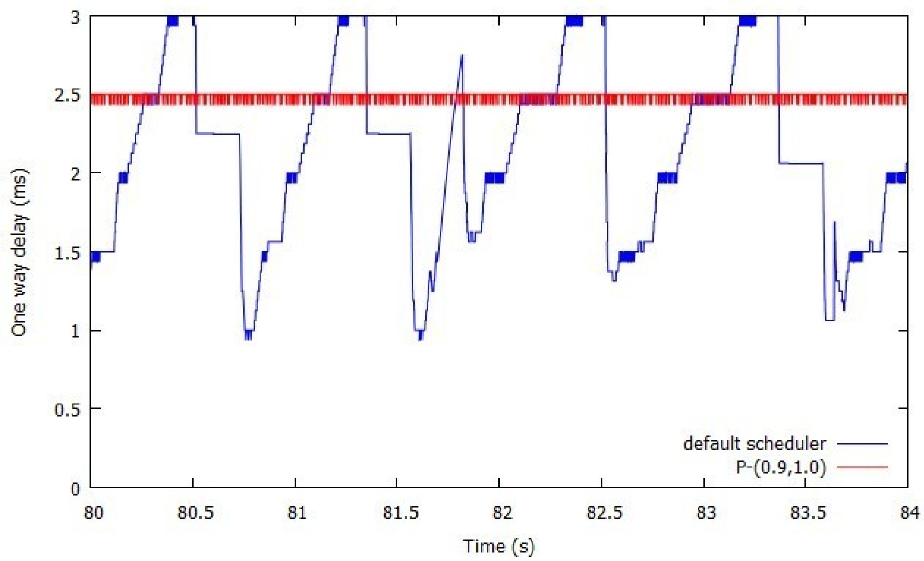

Figure 4.6: One Way Delay of Subflow 1 with Different Schedulers

decreases the cwnd so that the one way delay decreases. Consequently, the default scheduler is continually over-sending TCP-PDUs to a subflow and creating losses. We call this phenomenon *over-feeding*. Over-feeding is avoided by using P-(0.9,1.0) which prevents the number of outstanding TCP-PDUs from exceeding the estimated available path capacity so that the one way delay does not vacillate up and down.

When $\delta = 1.0$, our proposed scheduler gains roughly 14.5% throughput over the default scheduler. When $\delta > 1.0$, the number of retransmitted TCP-PDUs increases and over-feeding occurs. In our proposed scheduler, the default value of δ is 1.0. Therefore, our first hypothesis is confirmed by the results in Table 4.1. By alleviating over-feeding, our proposed scheduler can use network resources more efficiently.

4.6.2 Results with Cross Traffic

Our second hypothesis was that our proposed scheduler would be consistent with the aim of the MPTCP congestion control mechanism. That is, our proposed scheduler moves data away from congested links. This hypothesis is confirmed by the results in Table 4.2, which shows the results for the 150MB data transfer with cross traffic. When $\delta = 1.0$, our proposed scheduler gains roughly 6.5% throughput over the default scheduler. More importantly, our proposed scheduler sends more data to subflow 2 than the default scheduler since subflow 2 is less congested than subflow 1.

4.7 Discussions

Obviously, the efficiency of our proposed scheduler depends on γ and δ. If we can perfectly estimate subflows' available path capacities, over-feeding will be alleviated by setting $\delta = 1.0$. What should the proper value of γ be? On one hand, γ cannot be too large (i.e., approaching δ). Reconsider the problem reported in [22] (the topology is shown in Figure 3.8). Obviously, the srtt of subflow 1 is always shorter than that of subflow 2. If the default scheduler is used, subflow 1 is selected to send data whenever the subflow has available cwnd, while subflow 2 is only selected when subflow 1 has no available cwnd. If the available path capacity of subflow 1 $\geq$ the send buffer size, subflow 2 will only be used at the beginning of the data transfer. We call this phenomenon *biased-feeding*. If only subflow 1 is used, the maximum throughput is 10Mbps. While if both subflows are used, the maximum throughput is 21Mbps. Biased-feeding seriously decreases the parallelism of data transfer. If our proposed scheduler is used, a large γ also causes biased-feeding, since our proposed scheduler becomes the default scheduler when all subflows' $Occupied_i \leq \gamma$.

On the other hand, γ cannot be too small (i.e., approaching 0). Consider an MPTCP connection with two subflows where the total available path capacity of both subflows $\leq$ the send buffer size. Assume subflow 1 is established on a satellite link with longer delay and higher loss rate. Assume subflow 2 is established on a 3G path with shorter delay and lower loss rate. As γ approaches 0, more TCP-PDUs will be sent on subflow 1. Although biased-feeding is alleviated, at the MPTCP receiver side, the data arriving in-order on subflow 2 will not be delivered to the application layer since MPTCP level in-order data sent on subflow 1 has not arrived. The throughput using both subflows can be worse than only using subflow 2. This phenomenon is similar to the problem reported in [37]. This phenomenon decreases the throughput and can cause *GapAck-Induced send buffer blocking* [21].

Currently, in our proposed scheduler, the default values of γ and δ are 0.5 and 1.0, respectively. Ideally, the value of γ should be dynamically determined based on the characteristics (i.e., delay, packet loss and bandwidth) of subflows. Future work is proposed to investigate the dynamic determination of the value of γ.

4.8 Conclusion

To achieve successful deployment of MPTCP, one of the key problems is the cooperation of the scheduler and the congestion control mechanism. We admit the experiment setup was extremely limited (only one topology). However, these preliminary results demonstrate that the designs of both the scheduler and the congestion control mechanism cannot be separated, and it is important to design a scheduler which is consistent with the congestion control mechanism.

Chapter 5

USING ONE-WAY COMMUNICATION DELAY FOR IN-ORDER ARRIVAL MPTCP SCHEDULING

In this chapter, we use one-way communication delay of a TCP connection to design an MPTCP scheduler that transmits data out-of-order over multiple paths such that their arrivals are in-order. Our Linux implementation shows our proposed scheduler can reduce receive buffer utilization, and increase overall throughput when a small receive buffer size results in receive buffer blocking[1].

5.1 Motivations

Consider a hypothetical scenario of an MPTCP connection with two subflows as shown in Figure 5.1. Both subflows 1 and 2 have cwnd = 4, and the round trip time (RTTs) of subflows 1 and 2 are 200ms and 20ms, respectively. At a given time, assume 4 MPTCP-PDUs are outstanding on subflow 2, and MPTCP-PDU 5 is ready to be sent. The scheduler must decide on which subflow to send MPTCP-PDU 5?

According to both the default (used by the Linux MPTCP implementation) and our proposed schedulers in chapter 4, MPTCP-PDU 5 will be scheduled and sent on subflow 1 because subflow 2 has no available cwnd. For simplicity, assume the transmission delay of MPTCP-PDUs is ignored, and a subflow's one-way propagation delay is half of the subflow's RTT. It would take 10ms and 100ms for MPTCP-PDUs 1 to 4 and MPTCP-PDU 5 to arrive the receiver, respectively. The time between when MPTCP-PDU 1 is sent and when all 5 MPTCP-PDUs are delivered to the application layer would be 100ms, and a time interval of 90ms exists between the delivery time of MPTCP-PDUs 4 and 5 (Figure 5.2).

[1] These results have been published in [33] and [34].

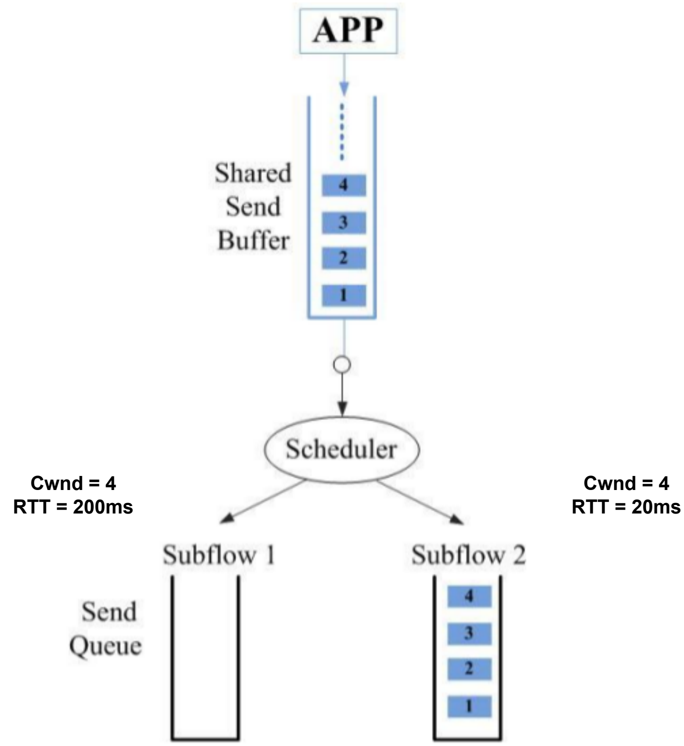

Figure 5.1: An MPTCP Connection with Two Subflows with Asymmetric RTTs

By applying a playout buffer to mitigate the jitter, this situation might not cause a problem for online streaming, where PDUs are sent out consistently. However, in some applications (e.g., online games), delay is sensitive, and PDUs are generated in short bursts. Relatively long time intervals between the delivery time of PDUs can occur periodically and negatively affect the user experience.

However, if MPTCP-PDU 5 was scheduled to subflow 2, that PDU just needs to wait for at most 1 RTT to be sent out. The time between when MPTCP-PDU 1 is sent and when all 5 MPTCP-PDUs are delivered to the application layer would be only 30ms and a time interval of 20ms exists between the delivery time of MPTCP-PDUs 4 and 5 (Figure 5.3).

Analogously, assume the application in Figure 5.1 has 34 MPTCP-PDUs ready

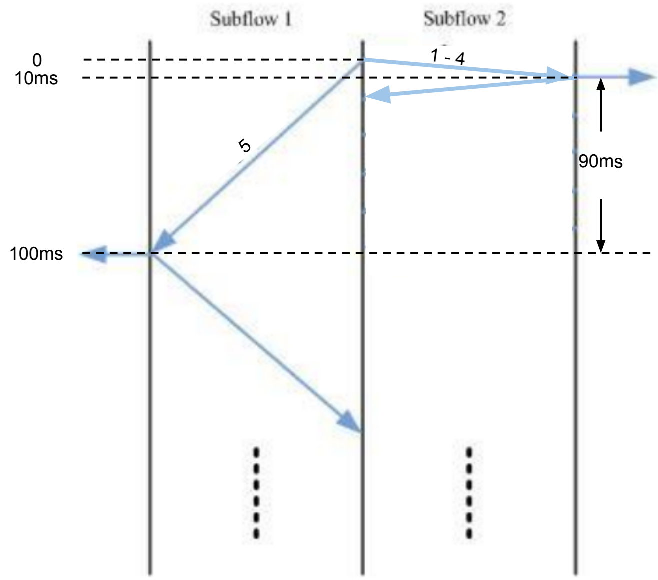

Figure 5.2: MPTCP Data Transfer (5 MPTCP-PDUs) with Two Subflows with Asymmetric RTTs

to be sent at the beginning. The timeline of data transfer with the default and our proposed schedulers in chapter 4 is shown in Figure 5.4. Although the total delivery delay of these 34 MPTCP-PDUs is 100ms, MPTCP-PDUs 9 to 34 received on subflow 2 have to reside in the MPTCP receive buffer and cannot be delivered until MPTCP-PDUs 5 to 8 are received on subflow 1. This situation causes an inefficient usage of the MPTCP receive buffer. Since an MPTCP receive buffer can be filled with out-of-order MPTCP-PDUs, receive buffer blocking occurs and the whole transmission stops eventually.

However, if MPTCP-PDUs 5 to 30 are sent on subflow 2 and MPTCP-PDUs 31 to 34 are sent on subflow 1, the total delivery delay remains 100ms, but the receive buffer only contains in-order MPTCP-PDUs at any given time.

Based on above example, if a scheduler can make all MPTCP-PDUs arrive in-order at the MPTCP receiver, both minimal jitter and more efficient usage of the MPTCP receive buffer can be achieved. However, the question is how difficult is it to design such a scheduler?

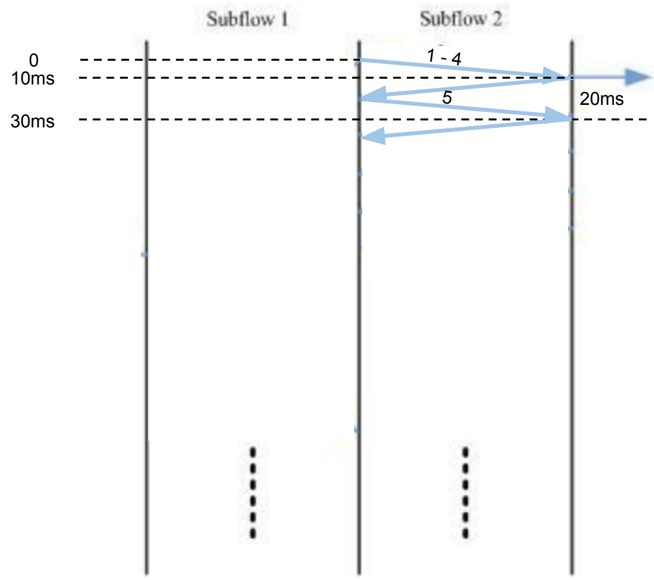

Figure 5.3: Improved MPTCP Data Transfer (5 MPTCP-PDUs) with Two Subflows with Asymmetric RTTs

5.2 Schedule MPTCP-PDUs to All Established Subflows

Reconsider the task of a scheduler [32]. Whenever an MPTCP sender wants to send data, the sender needs to make three decisions. First, which subflow(s) can be used (i.e., has available cwnd) to send data? Second, if several subflows have available cwnds, which subflow should be chosen? Third, after selecting a subflow, how much data should be scheduled to it? The third decision concerns the granularity of the allocation. For the purpose of simplicity, we just temporarily assume each subflow has the same maximum segment size (MSS), and an MPTCP sender allocates one MSS at a time in step 3. We will discuss the implementation problems in the situation where subflows have different MSS at the end of this chapter. Now, we focus on the scheduler's first two decisions.

Note that, all existing schedulers (including the default) only schedule MPTCP-PDUs to subflows with available cwnds. That is, at any given time, only subflows with available cwnd can be used to schedule data. This default policy is why MPTCP-PDUs 5 to 8 are scheduled to subflow 1 in Figure 5.4. However, to achieve in-order arrival

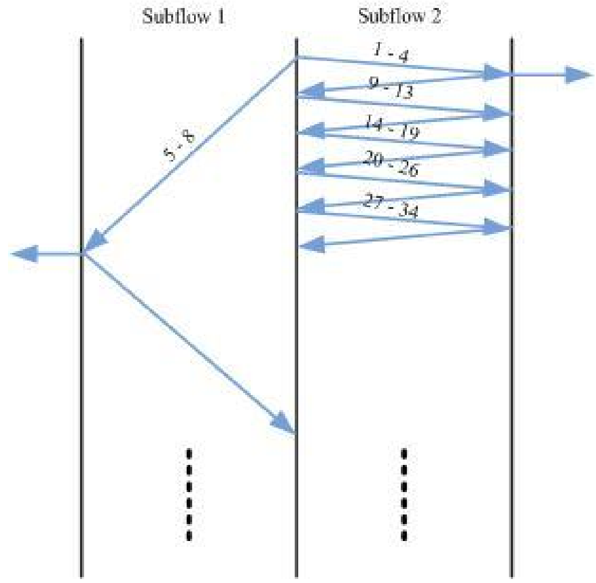

Figure 5.4: MPTCP Data Transfer (32 MPTCP-PDUs) with Two Asymmetric Subflows

scheduling, in answering the first decision, **all established subflows can be used to schedule data, even those with no available cwnd**.

To make MPTCP-PDUs arrive in-order at the receiver side, the answer to the second decision is derived as follows. We define the time range between when an MPTCP-PDU is scheduled to a subflow and when that PDU arrives in-order at the subflow's receive buffer as **delivery delay** (DeD). DeD_j^i is the delivery delay of MPTCP-PDU i if scheduled to subflow j. Note that, "MPTCP-PDU i is scheduled to subflow j" does not mean "MPTCP-PDU i can be sent immediately on subflow j", since subflow j may have no available cwnd currently. At any given time, if MPTCP-PDU i is ready to be scheduled and the scheduler knows DeD_i for all subflows, **MPTCP-PDU i will be scheduled to the subflow with the smallest** DeD_i. Each subflow will send out scheduled MPTCP-PDUs whenever the subflow has available cwnd. With this policy, if the network does not drop or reorder PDUs, all MPTCP-PDUs will arrive 'perfectly' in-order at the MPTCP receiver. Note: although MPTCP-PDUs are scheduled in-order, they can be sent out out-of-order.

5.3 One-way Communication Delay

One problem is "how to calculate $\mathrm{DeD}_{\mathrm{i}}^{\mathrm{j}}$". $\mathrm{DeD}_{\mathrm{i}}^{\mathrm{j}}$ comprises two parts: (i) the time which MPTCP-PDU i spends in subflow j's send buffer, and (ii) the time range between when MPTCP-PDU i is sent out (for the first time) and when it is received in-order at subflow j's receive buffer. Obviously, part (i) is 0 if subflow j has available cwnd when MPTCP-PDU i is scheduled. Part (ii) actually represents how long it takes for a subflow sender to truly *communicate* its data to the receiver, and is denoted **One-Way Communication Delay** (CommD). CommD is not the same as traditional one-way delay [52, 53, 54] which measures propagation, transmission and queueing delays without taking into account delays due to retransmissions.

Please note, CommD is the characteristic of a subflow. However, DeD is the delivery delay of a specific MPTCP-PDU on a specific subflow. Suppose MPTCP-PDU i is scheduled to subflow j. If subflow j has available cwnd, $\mathrm{DeD}_{\mathrm{j}}^{\mathrm{i}} = \mathrm{CommD}_{\mathrm{j}}$. Otherwise, $\mathrm{DeD}_{\mathrm{j}}^{\mathrm{i}} = \mathrm{Time}_{\mathrm{spent\_in\_sndbuf\_j}}^{\mathrm{i}} + \mathrm{CommD}_{\mathrm{j}}$.

We introduce how to measure CommD of a subflow in this section and how to compute $\mathrm{Time}_{\mathrm{spent\_in\_sndbuf}}$ in section 5.4.

A negative aspect of using a subflow's CommD is its difficulty to measure in practice since the metric is distributed: the start and stop times of the CommD interval occur on different machines. But since the start and end hosts are identical for all subflows, in our proposed measurement scheme, the end point clocks need not be synchronized. A scheduler only needs to know which subflow has the shortest CommD not the specific values of all subflows. The scheduler can easily measure a CommD' which is defined as $\mathrm{CommD} + \mathrm{C}$. Here, C is the time difference between the end host clocks.

Let us present a hypothetical example to demonstrate how to measure the CommD' of a subflow in MPTCP. Similar to TCP's measurement of RTT, only one CommD' measurement sample can be in progress at any time. We denote $\mathrm{CommD}_{\mathrm{i}}^{\mathrm{k}}$ and $\overline{\mathrm{CommD}_{\mathrm{i}}}$ as the jth measured sample and smoothed average CommD' of subflow i, respectively. Here, 'sample' and 'smoothed' have the analogous meaning as those in

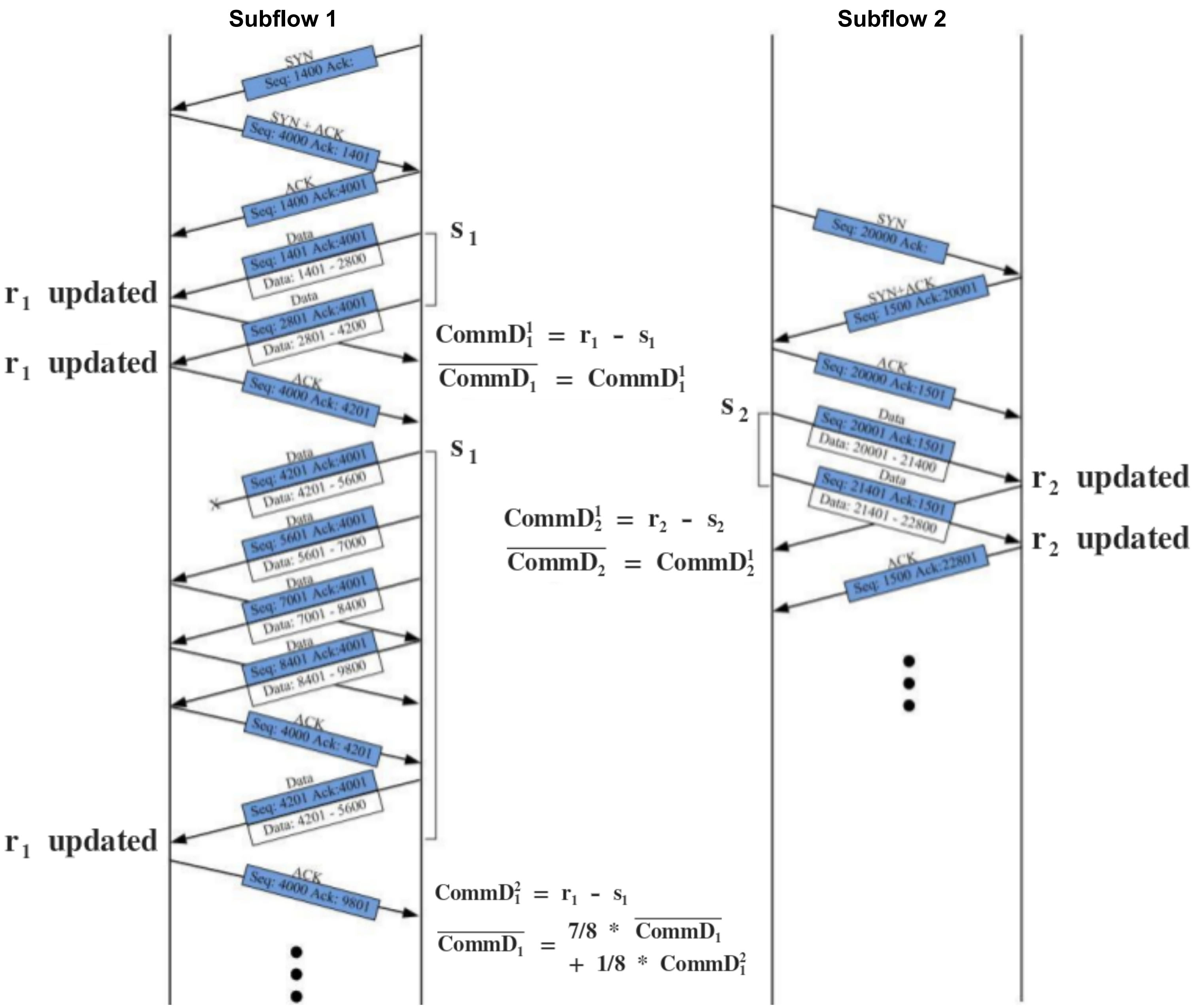

Figure 5.5: Example of CommD Measurement for MPTCP

traditional TCP RTT measurement. s_i is the time when an MPTCP-PDU (seq: S_s - S_e) is transmitted for the first time, and is recorded by subflow i's sender as the start time of a sample measurement. Subflow i's receiver constantly updates a variable r_i to be the time when a PDU is received in-order. The receiver echos the latest value of r_i to the sender in the acknowledgments. The sender pairs the r_i in the first received acknowledgment (with acknowledgment number $\geq S_e$) with current s_i. Figure 5.5 shows an MPTCP connection with two subflows. Two samples are collected for subflow 1, and only one sample is collected for subflow 2.

1. Subflow 1's first CommD′ measurement starts when TCP-PDU (Sub seq: 1401 - 2800) is sent, and s_1 is updated correspondingly. No CommD′ measurement starts for the next TCP-PDU (Sub seq: 2801 - 4200) because a measurement is already in progress.

2. Subflow 1's receiver updates r_1 to be the time when TCP-PDU (Sub seq: 1401 - 2800) is received. Then, r_1 is included in ACK (ack: 2801).

3. When ACK (ack: 2801) arrives at subflow 1's sender, it computes $\mathrm{CommD}_1^1 = r_1 - s_1$. Also, $\overline{\mathrm{CommD}_1} = \mathrm{CommD}_1^1$. The sender then clears the value of s_1 to indicate that no measurement is in progress.

4. s_1 is updated to be the time when TCP-PDU (Sub seq: 4201 - 5600) is sent out. In this example, this TCP-PDU is presumed lost.

5. Subflow 1's receiver does not update r_1 until the retransmission (either by time out or fast retransmission) of the lost TCP-PDU is ultimately received.

6. Subflow 1's sender does not compute CommD_1^2 until the arrival of ACK (ack: 9801) which is the first received acknowledgment with ack ≥ 5600. Also,
$\overline{\mathrm{CommD}_1} = (7/8 * \overline{\mathrm{CommD}_1}) + (1/8 * \mathrm{CommD}_1^2)$.

7. Similarly, subflow 2's first CommD′ measurement starts when TCP-PDU (Sub seq: 20001 - 21400) is sent out. $\overline{\mathrm{CommD}_2} = \mathrm{CommD}_2^1 = r_2 - s_2$.

Note that, the accuracy of CommD′ measurement is influenced by not only delayed acknowledgment but also acknowledgment losses. CommD is an one-way delay and also accurately accounts for losses of a subflow.

5.4 Time Spent in the Send Buffer

Now, let us see how a scheduler can calculate the time ($\text{Time}^{\text{i}}_{\text{spent\_in\_sndbuf\_j}}$) which MPTCP-PDU i spends in the send buffer if MPTCP-PDU i is scheduled to subflow j. The send buffer of subflow j comprises (i) newly scheduled data waiting to be transmitted for the first time, and (ii) data that is outstanding (i.e., already transmitted at least once and awaiting to be cum-acked). In Figure 5.6, these data are called Not_yet_sent and Outstanding, respectively.

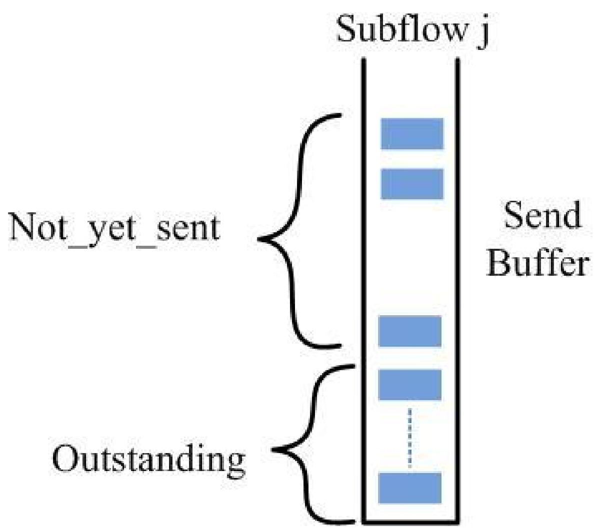

Figure 5.6: Send Buffer of a Subflow

For subflow j, $\text{Outstanding}_{\text{j}}$ can be $\leq \text{Cwnd}_{\text{j}}$ (assume subflow j is not in fast recovery state). Therefore, some PDUs can be sent immediately, then:

$$\text{Number\_of\_packets\_can\_be\_sent}_{\text{j}} = \text{Cwnd}_{\text{j}} - \text{Outstanding}_{\text{j}} \tag{5.1}$$

The number of RTTs which MPTCP-PDU i needs to wait to be sent out is:

$$\text{Number\_of\_RTTs\_wait}^{\text{j}}_{\text{i}} = \left\lfloor \frac{\text{Not\_yet\_sent}_{\text{j}} - \text{Num\_packets\_can\_be\_sent}_{\text{j}}}{\text{Cwnd}_{\text{j}}} \right\rfloor \tag{5.2}$$

Therefore, the time which MPTCP-PDU i spends in subflow j's send buffer is:

$$\mathrm{Time_{spent\_in\_sndbuf}} = \mathrm{Number\_of\_RTTs\_wait}_i^j * \mathrm{RTT_j} \tag{5.3}$$

Unlike the CommD measurement, the calculation of $\mathrm{Time_{spent\_in\_sndbuf}}$ does not account for losses. Accounting losses in the calculation of $\mathrm{Time_{spent\_in\_sndbuf}}$ is beyond the scope of this dissertation and will be included in our future work.

5.5 Two Designs of In-order Arrival Scheduling

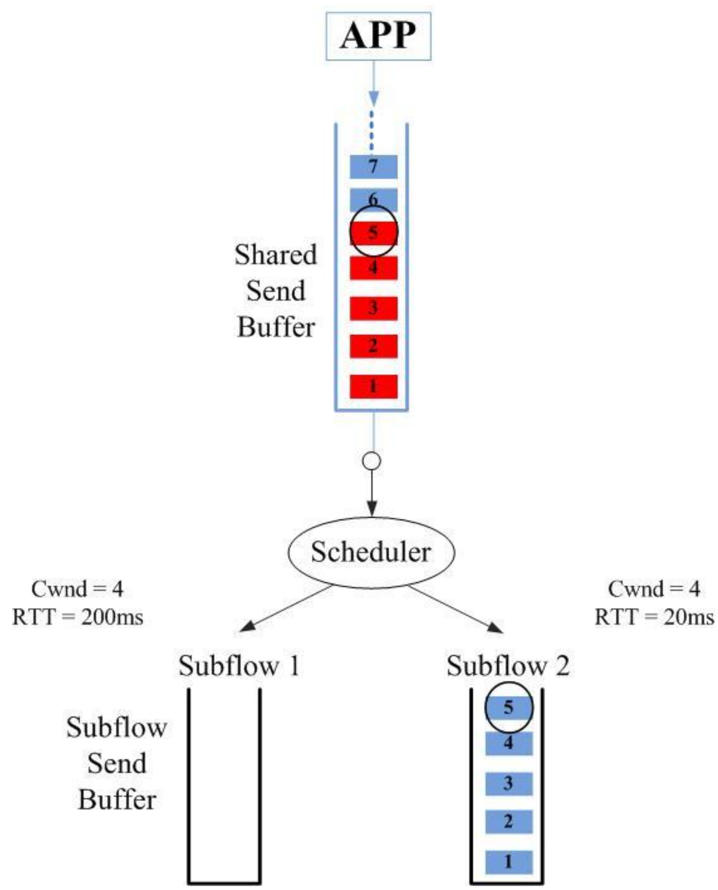

Figure 5.7: Design 1: MPTCP-PDUs are Always Scheduled In-order

Now, we know the delivery delay of MPTCP-PDU i if it is scheduled to subflow j is:

$$\mathrm{DeD}_j^i = (\mathrm{Number\_of\_RTTs\_wait}_i^j * \mathrm{RTT}_j) + \mathrm{CommD}_j \tag{5.4}$$

Reconsider the target of our scheduler: to transmit MPTCP-PDUs on different subflows possibly out-of-order so that they arrive in-order at the MPTCP receive buffer. To achieve this target, two problems need to be solved: one is how to select the next MPTCP-PDU to be scheduled, and the other is on which subflow should this selected MPTCP-PDU be transmitted. We have the answer to the second problem: If MPTCP-PDU i is ready to be scheduled, a scheduler needs to compute DeD_j^i for each subflow j, and then schedule MPTCP-PDU i to the subflow with the shortest DeD_j^i.

A quick, yet inefficient answer to the first problem (Design 1) is "just select the next not yet scheduled MPTCP-PDU in the MPTCP send buffer". Assume MPTCP-PDU i is selected and scheduled to subflow j. MPTCP-PDU i is encapsulated in a TCP-PDU and placed in subflow j's send buffer. At a given time, all scheduled MPTCP-PDUs will have two copies. As shown in Figure 5.7, although MPTCP-PDU 5 cannot be sent out immediately, a second copy is placed in subflow 2's send buffer. However, for the default MPTCP scheduler, only in-flight MPTCP-PDUs have two copies. Please note, this first design always schedules MPTCP-PDUs in-order. Doing so brings a result: a more efficient usage of the receive buffer which incurs a less efficient usage of the send buffer. We cannot say this solution is beneficial.

A preferable solution (Design 2) would be to only maintain two copies of in-flight MPTCP-PDUs, and still achieve in-order arrival. We need to modify the quick answer to the first problem to be "select an unscheduled MPTCP-PDU which can be sent out now". For example, MPTCP-PDU i is the next as yet unscheduled MPTCP-PDU, and subflow j has the shortest DeD_j^i. If subflow j has available cwnd, MPTCP-PDU i is scheduled and sent out on subflow j. If subflow j has no available cwnd, MPTCP-PDU i is 'assumed' to be scheduled (what we refer to as 'dummy scheduling') to subflow j but will not be copied just yet to subflow j's send buffer.

The scheduler continues to consider MPTCP-PDU $i + 1$ until an unscheduled

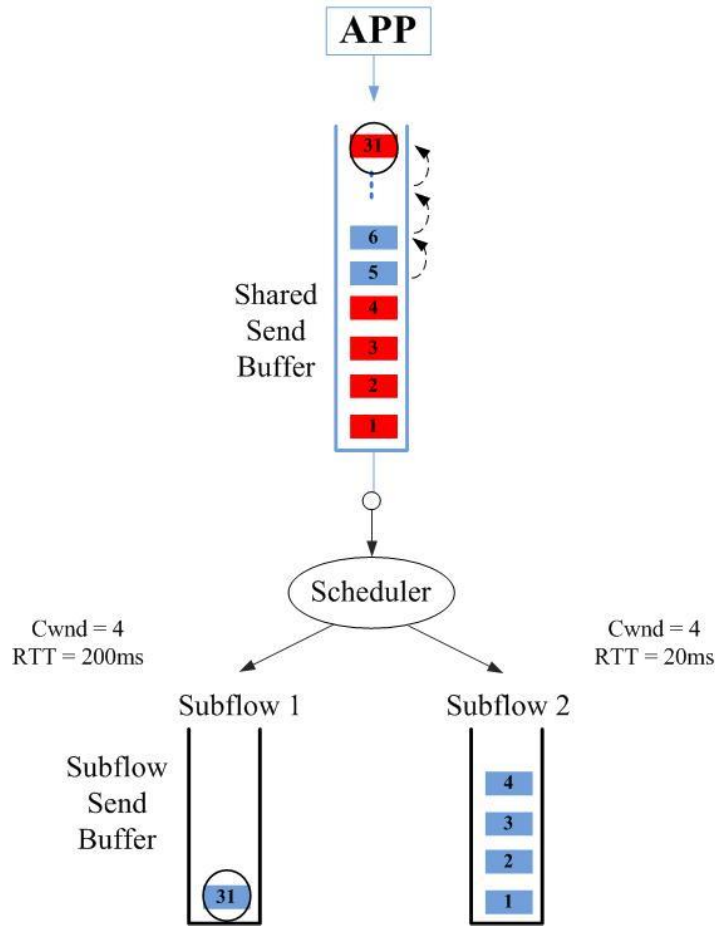

Figure 5.8: Design 2: MPTCP-PDUs can be Scheduled Out-of-order

MPTCP-PDU k is found, and a subflow l has both available cwnd and the shortest $\mathrm{DeD}_{\mathrm{l}}^{\mathrm{k}}$. Then, MPTCP-PDU k is copied to the send buffer of subflow l and transmitted immediately. Dummy scheduling is necessary to maintain the correctness of the DeD calculation. As shown in Figure 5.8, the scheduler continues searching in the send buffer until MPTCP-PDU 31 (which is as yet unscheduled and can be sent out immediately on subflow 1) is found, then MPTCP-PDU 31 is copied to subflow 1's send buffer. Compared to Design 1, Design 2 schedules MPTCP-PDUs **out-of-order** and avoids duplicates of MPTCP-PDUs. In the next section, we describe the implementation pseudo-code for Design 2.

5.6 Implementation

We implemented this scheduler in Linux kernel based on the Linux MPTCP [35]. The following functions are related to the scheduler:

`mptcp_next_segment`: selects the next MPTCP-PDU to be scheduled.

`get_available_subflow`: selects a subflow to transmit a selected MPTCP-PDU.

`mptcp_write_xmit`: transmits a selected MPTCP-PDU on a selected subflow.

Below, we present the pseudo-code of modified `mptcp_next_segment` to achieve scheduling MPTCP-PDUs out-of-order. The following variables are used:

`skb`: a pointer to an MPTCP-PDU;

`meta_sk`: the MPTCP level socket structure;

`subsk(subtp)`: a subflow level socket(tcp socket) structure;

`path_mask`: a variable recording an MPTCP-PDU has been scheduled to which subflow(s);

`path_index`: index of a subflow;

`pre_sched_path_index`: a variable indicating which subflow should an MPTCP-PDU be scheduled to;

`DeD`: delivery delay of an MPTCP-PDU;

`CommD`: communication delay of a subflow;

`dummy_sched_packets`: number of MPTCP-PDUs which are assumed to be scheduled to a subflow.

Pseudo code

```
skb = tcp_send_head(meta_sk);        ▷ get 1st MPTCP-PDU in send buffer

for each subtp do    ▷ initialize the number of 'dummy' scheduled MPTCP-PDUs
   subtp->dummy_sched_packets = 0;
end for

while skb is not NULL do
```

```
subsk = get_available_subflow(meta_sk);
if subsk is NULL then                              ▷ no subflow is available
    return NULL;                                   ▷ delay the scheduling
end if

if TCP_SKB_CB(skb)->path_mask then ▷ this MPTCP-PDU has been scheduled
    goto next_skb;
end if

min_DeD = 0xFF;            ▷ initialize to be the maximal 8-bit unsigned int
min_DeD_subtp = NULL;                       ▷ initialize the selected subflow
for each subtp do
    if subtp has no available cwnd then
        DeD = (subtp->dummy_sched_packets / subtp->snd_cwnd + 1) *
        subtp->srtt + subtp->CommD;                          ▷ equation 5.4
    else if subtp has available cwnd then
        DeD = subtp->CommD;  ▷ DeD equals CommD when has available cwnd
    end if

    if DeD < min_DeD then                 ▷ select subflow with shortest DeD
        min_DeD = DeD;
        min_DeD_tp = subtp;
    end if
end for

if min_DeD_tp has available cwnd then       ▷ schedule selected MPTCP-PDU
    TCP_SKB_CB(skb)->pre_sched_path_index = min_DeD_tp->mptcp->path_index;
    return skb;
else
```

```
        subtp->dummy_sched_packets++;                      ▷ 'dummy' scheduling
    end if

    next_skb:
    if tcp_skb_is_last(meta_sk, skb) then                  ▷ reach send buffer end
        skb = NULL;
    else
        skb = tcp_write_queue_next(meta_sk, skb);          ▷ check next MPTCP-PDU
    end if
end while
```

This modified `mptcp_next_segment` returns both the unscheduled MPTCP-PDU and the corresponding subflow. Therefore, `mptcp_write_xmit` just needs to transmit the returned MPTCP-PDU on the subflow.

5.7 Results of In-order Arrival Scheduling

5.7.1 Test-bed Topology

Our test-bed, depicted in Figure 5.9, consists of two Cisco Linksys routers and two laptops running the MPTCP (v0.88) kernel. We use Opportunistic Linked Increases Algorithm (OLIA) as the default congestion control mechanism [47]. Both laptops are multihomed by using their tethered Ethernet interface and a Cisco USB Ethernet adapter. An MPTCP connection is established between the two laptops. Subflow 1 is established over the two tethered Ethernet interfaces, while subflow 2 is established between the two Cisco USB Ethernet adapters. Each Cisco USB Ethernet adapter comes with a small internal buffer that can only queue up to 3 PDUs, thus the intermediate buffer size of subflow 2 is smaller than that of subflow 1. If the intermediate buffers of both subflows are full, the RTT of subflow 2 will be shorter than that of subflow 1. We use FTP to generate MPTCP traffic to confirm the in-order arrival of our proposed scheduler.

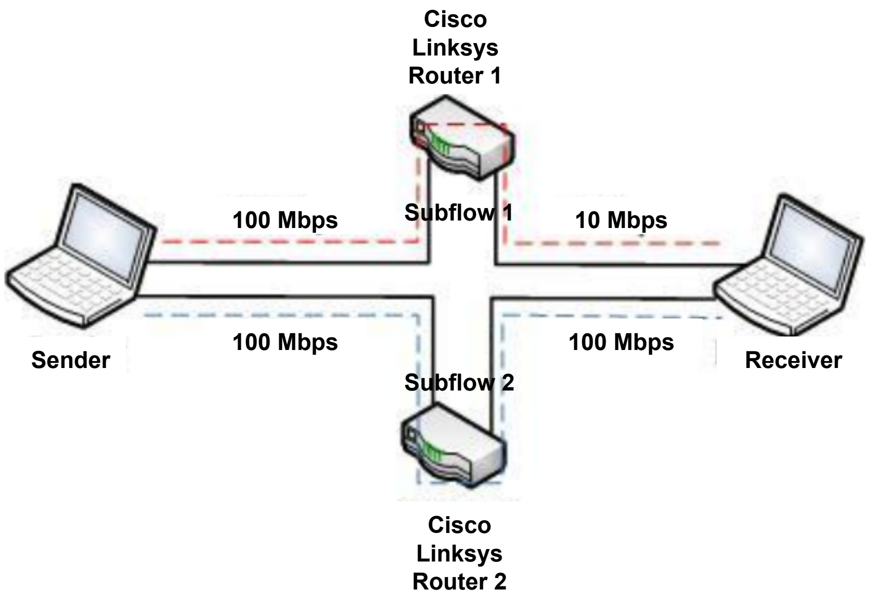

Figure 5.9: Test-bed Topology

5.7.2 Receive Buffer Usage

We hypothesized that our proposed scheduler would occupy less receive buffer space than the default scheduler. Figure 5.10 shows the size of occupied receive buffer size for the default and our proposed schedulers) during the time interval from 20s to 105s of a data transfer. The default scheduler can occupy as much as 342KB which is 38% of the entire allocated receive buffer space. The default scheduler reaches the steady state after 95s and occupies 83KB in average. Our proposed scheduler always occupies 25KB, roughly 2.8% of the available buffer space. Our hypothesis is confirmed and our proposal can reduce the usage of the receive buffer for up to 35.2% for this topology.

5.7.3 Throughput with Reduced Receive Buffer

In an MPTCP data transfer, if the sender always has enough traffic to fill all subflows (i.e., $\mathrm{Sbuf} \geq \sum_{\mathrm{all\ subflow\ i}} \bar{\mathrm{w}}_\mathrm{i}$ (where $\bar{\mathrm{w}}_\mathrm{i}$ is the average cwnd of subflow i in

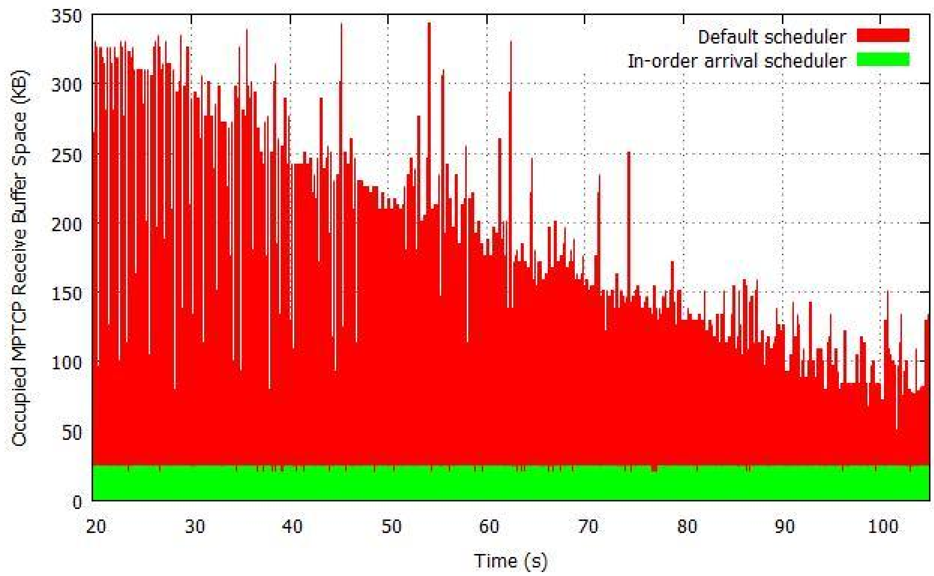

Figure 5.10: Receive Buffer Usage

Table 5.1: Throughput Comparison with Reduced Receive Buffers

Receive Buffer (KB)	889	796	707	619	530	442	354	265	177
Default Scheduler (MBps)	1.47	1.47	1.47	1.47	1.47	1.47	1.21	1.20	1.14
In-order Arrival Scheduler (MBps)	1.47	1.47	1.47	1.47	1.47	1.47	1.47	1.47	1.47

equilibrium status)) and the receiver always has enough space to accommodate out-of-order MPTCP-PDUs, the scheduler cannot influence the throughput. However, when the receive buffer decreases, we hypothesize our proposed scheduler will provide greater throughput than the default scheduler. Table 5.1 shows the throughput achieved by both schedulers for a variety of receive buffer sizes. Each shown throughput value in the table is the average throughput of 50 data transfers with specified scheduler under specified receive buffer size. We can see the throughput of the default scheduler starts decreasing when the receive buffer is 354KB or smaller, while that with our proposed scheduler remains steady even the receive buffer is reduced to only 177KB.

5.8 Limitations

5.8.1 Subflows with Different MSS

These experimental results are promising. However, our current implementation has a strong assumption that all subflows have the same Maximum Segment Size (MSS). If the subflows of an MPTCP connection have different MSS, the scheduling problem becomes more complicated.

The simplest method is a scheduler always schedules MPTCP-PDUs of the smallest MSS to subflows. Obviously, this method is inefficient. We need to find out other possible solutions.

In the MPTCP Linux implementation, each `skb` (detailed introduction of skb can be found in Chapter 2) in the MPTCP send buffer is allocated with data section size of minimal MSS of all subflows. Please note, skbs and MPTCP-PDUs do not necessarily have an one-to-one correspondence. In other words, an skb can represent multiple MPTCP-PDUs, and an MPTCP-PDU may comprise several skbs. As a special case, when the subflows have the same MSS, an skb exactly represents an MPTCP-PDU.

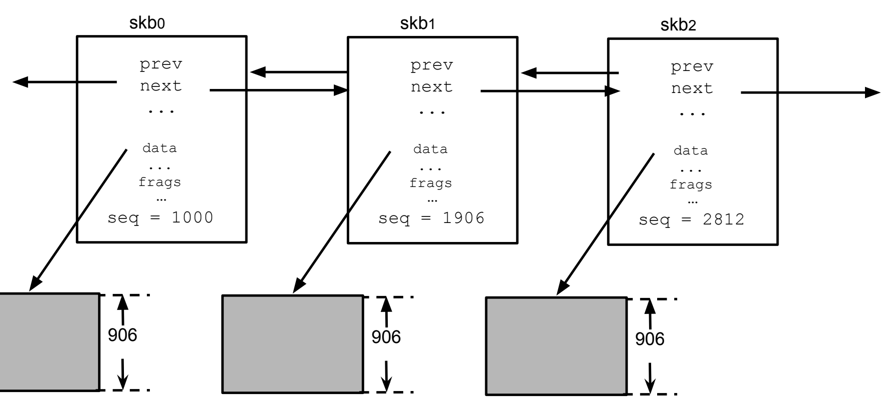

Figure 5.11: `skb`s in the MPTCP Send Buffer

Let us consider an example: an MPTCP connection is established with two subflows. Subflow 1 has an MSS = 1400 bytes and subflow 2 has an MSS = 906 bytes. Figure 5.11 shows the skbs (each has a data section of size 906 bytes) in the MPTCP send buffer. At the shown time, assume subflow 1 has no available cwnd and subflow 2 has available cwnd = 2.

The size of a scheduled MPTCP-PDU depends on the corresponding subflow's MSS. Assume MPTCP-PDU whose Data Sequence Number (DSN) starts from 1000 has the shortest DeD on subflow 1. This scheduling is delayed since subflow 1 has no available cwnd. The start DSN of next not yet scheduled MPTCP-PDU is 2400, because the MSS of subflow 1 is 1400 bytes. Assume this MPTCP-PDU has the shortest DeD on subflow 2. Since subflow 2 has available cwnd, MPTCP-PDU 2400 - 3305 is scheduled and sent out. The scheduler needs to maintain a list (`scheduled_DSN_list`) to record scheduled blocks of DSNs.

The pseudo-code in section 5.6 needs to be modified:

Modified pseudo code

Start from DSN of the MPTCP send buffer's left edge

Find the start DSN ($\mathrm{DSN}_{\mathrm{next}}$) of next not yet scheduled MPTCP-PDU from the `scheduled_DSN_list`

`min_DeD = 0xFF;` ▷ initialize to be the maximal 8-bit unsigned int

`min_DeD_subtp = NULL;` ▷ initialize the selected subflow

for each `subtp` **do**

  if `subtp` has no available cwnd **then**

    `DeD = (subtp->dummy_sched_packets / subtp->snd_cwnd + 1) * subtp->srtt + subtp->CommD;` ▷ equation 5.4

  else if `subtp` has available cwnd **then**

    `DeD = subtp->CommD;` ▷ DeD equals CommD when has available cwnd

  end if

```
    if DeD < min_DeD then                          ▷ select subflow with shortest DeD
        min_DeD = DeD;
        min_DeD_tp = subtp;
    end if
end for

if min_DeD_tp has available cwnd then
    schedule MPTCP-PDU (DSN_next to DSN_next + MSS_min_DeD_tp) to min_DeD_tp
else if min_DeD_tp has no available cwnd then
    min_DeD_tp->dummy_sched_packets++;             ▷ 'dummy' scheduling
    DSN_next += MSS_min_DeD_tp        ▷ increase DSN_next by this subflow's MSS
else
    return
end if
```

However, this solution has a problem if the network situation changes. Take the above example, after MPTCP-PDU 2400 - 3305 is sent out on subflow 2, assume MPTCP-PDU whose DSN starts from 1000 now has the shortest DeD on subflow 2. Since subflow 2 still has available cwnd (= 1), MPTCP-PDU 1000 - 1905 is scheduled and sent out. As shown in Figure 5.12, MPTCP-PDUs 1000 - 1905 and 2400 - 3305 have been scheduled and sent out (shaded rectangles). However, duplicate data will be sent when the MPTCP-PDU whose DSN starts from 1906 is scheduled (to no matter which subflow). If subflow 1 is selected, MPTCP-PDU 1906 - 4305 would be sent with 906 bytes duplicated. If subflow 2 is selected, MPTCP-PDU 1906 - 2811 would be sent with 412 bytes duplicated.

5.8.2 Only Accounting for Losses in CommD

If our proposed scheduler can 'perfectly' make MPTCP-PDUs arrive in-order at the receiver and the application always consumes all in-order MPTCP-PDUs, the

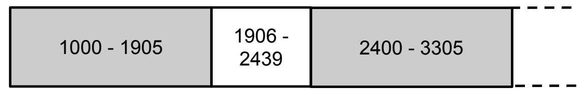

Figure 5.12: Blocks of DSNs in the MPTCP Send Buffer

occupied receive buffer size in Figure 5.10 should be 0. Obviously, the efficiency of our proposed scheduler depends on the accuracy of the algorithm to estimate DeD_j^i (equation 5.4). However, our proposed algorithm to calculate DeD_j^i only accounts for losses in computing CommD_j^i but not $\mathrm{Time}^i_{\mathrm{spent\_in\_sndbuf\_j}}$.

From equations 5.2 and 5.3, a scheduler has at any given moment accurate information for Not_yet_sent and Num_packets_can_be_sent. However, a subflow's RTT keeps changing as the intermediate queues on the subflow's path expand or shrink. A subflow's cwnd increases when acknowledgments come back and decreases when losses occur. Using a subflow's current RTT and cwnd to determine the future scheduling would be inaccurate when losses occur, or congestion changes on the network path. Extensive experiments are needed to determine under what circumstances (e.g., loss rate ranges of subflows) our proposed scheduler will outperform the default one.

This investigation is just a beginning in moving in-order arrival scheduling from theory to practice, and future research is needed to achieve a complete solution.

Chapter 6

PRIOR COLLABORATIVE RESEARCH

Prior to the research contributions of this dissertation, I was involved with deriving an appropriate initial dictionary for SPDY to compress HTTP headers using the dictionary data compression method zlib. I have also been involved with a project to support past PhD student Nasif Ekiz. The activity extended the wireshark flow graph to help analyzing TCP flows. This chapter presents these two contributions.

6.1 Methodology to derive SPDY's Initial Dictionary

Google is proposing a new application-layer protocol SPDY (pronounced "SPeeDY") as a way of making browsing the Internet faster [55]. HTTP is the application level protocol providing basic request/reply semantics. Unfortunately, HTTP was not designed to minimize latency. One particular HTTP feature, the use of uncompressed request and reply headers, inhibits optimal performance.

SPDY compresses HTTP request and reply headers using zlib [56], a widely-used dictionary-based compression method. Zlib is a lossless data compression library which provides in-memory compression and decompression functions. The current zlib library defines methods 'deflate' and 'inflate' which provide compression and decompression, respectively.

Several design issues were considered in developing a methodology to derive an initial dictionary:

Size of the initial dictionary: The current implementation of zlib's deflate method will use at most the window size (number of bytes that can be compressed at the same time) minus 262 bytes of the provided dictionary. The number of bits of window size in SPDY is 11, thus the size of initial dictionary should not exceed 1786 bytes (i.e., $2^{11} - 262$). This size limitation prevents including all HTTP specification keywords in the initial dictionary. Some specified HTTP keywords are rarely used in practice. Thus in our methodology, only keywords regularly used in practice should be included. If and when HTTP keywords change in popularity in the future, a revised initial dictionary can be derived at that time.

Popular websites/browsers not included: A conscious decision was made not to include specific popular websites (e.g., facebook, yahoo, Google) or browsers (e.g., firefox), since we do not want to bias the SPDY protocol against new companies/software trying to enter the future marketplace.

Frequency to update the initial dictionary: An update of zlib's initial dictionary for SPDY requires modifying both peer end points (client browser and server), so changes are problematic. If the initial dictionary is not permitted to change or can only change infrequently (e.g., every few years), we want today's initial dictionary to be appropriate in the future. An obvious example is that the year frequently appears in HTTP headers, so 2014 appears frequently in today's HTTP headers. In two years, however, 2016 will appear frequently and perhaps 2014 not at all. To allow for our initial dictionary to be applicable for several years, we exclude time-dependant keywords (e.g., 2014) from the initial dictionary.

The methodology used to derive SPDY's initial dictionary for zlib compression was as follows:

1. Collect HTTP reply headers (in ASCII form) from the main page of the top 1000 websites based on [57] as our training set of HTTP replies. Our training set of HTTP requests was provided by Mike Belshe (Google). These two training sets can be found in [58], [59], respectively.

2. Use punctuation (i.e., blank, comma, newline, semicolon) to parse the headers in both training sets into keywords.

3. Calculate a weight for each keyword in HTTP reply headers by considering the frequency of page views for the top 1000 websites (6th column in [57]). For example, suppose we have three HTTP reply headers (with HTTP version 'HTTP/1.1') which are

replies from the main pages of facebook.com, youtube.com and yahoo.com, respectively. Then the weight of the keyword 'HTTP/1.1' is sum of the frequency of page views of these three websites. Calculate the count of each keyword in HTTP request headers based on the number of appearances of the keyword. For example, suppose we have three HTTP request headers (with method 'GET'), then the count of keyword 'GET' is three.

4. Build an initial dictionary for HTTP replies based on the weight of each keyword in HTTP reply headers. Build an initial dictionary for HTTP requests based on the count of each keyword in HTTP request headers. Since the SPDY designers preferred to have just a single dictionary for both headers and replies, we then concatenate these two dictionaries. For HTTP header field names and some fixed length values (e.g., HTTP/1.1, 200 OK), we include the 32-bit length prefix before the word. (Note: a separate initial dictionary optimized for either HTTP requests or replies would likely provide better compression, but would in turn make SPDY's implementation more complex.)

5. Add some 'known' common non keywords in the dictionary. Example words include HTTP status codes (e.g., 100, 101, 201), months (e.g., Jan, Feb) and days (e.g., Mon, Tue). These words can be called 'metadata', which repeat often in HTTP headers and are unlikely to change in the future.

After applying our methodology, and by using our proposed initial dictionary, an additional 8% compression of the first HTTP request header and an additional 15% compression of first HTTP reply header on a SPDY connection is gained over SPDY's current default initial dictionary. For the 2nd, 3rd, and further HTTP request (or reply) headers transmitted over the same SPDY connection, compression using our proposed initial dictionary is practically identical to that achieved by SPDY's default initial dictionary. This result suggests zlib's adaptive dictionary evolves to roughly the same state after compressing the first HTTP header regardless of the initial dictionary.

6.2 Wireshark Extensions

Reneging occurs when a data receiver first selectively acknowledges data, and later discards that data from the receiver buffer before delivery to the receiving application. When Nasif Ekiz was analyzing TCP flows to find reneging instances, the flow graph (which can display TCP-PDUs in a timeline (Figure 6.1)) in wireshark was a good tool. However, the default flow graph has these drawbacks:

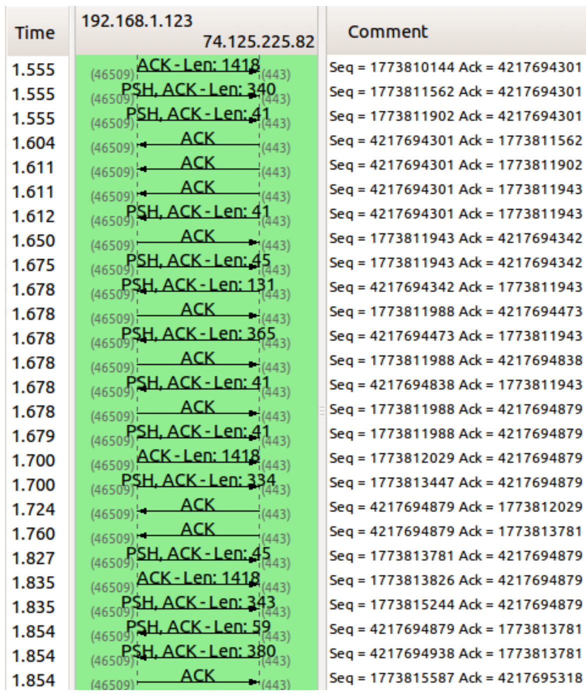

Figure 6.1: Default Flow Graph in Wireshark

Displaying information of both directions at the same column: The default flow graph displays ‘Seq’ and ‘Ack’ information of both directions in the same ‘Comment’ column, which is really unclear and confusing.

TCP SACK information not displayed: TCP SACK information is crucial for analyzing reneging instances, but the default flow graph does not display TCP SACKs.

Based on these drawbacks, I extended the flow graph in wireshark to support following features (Figure 6.2):

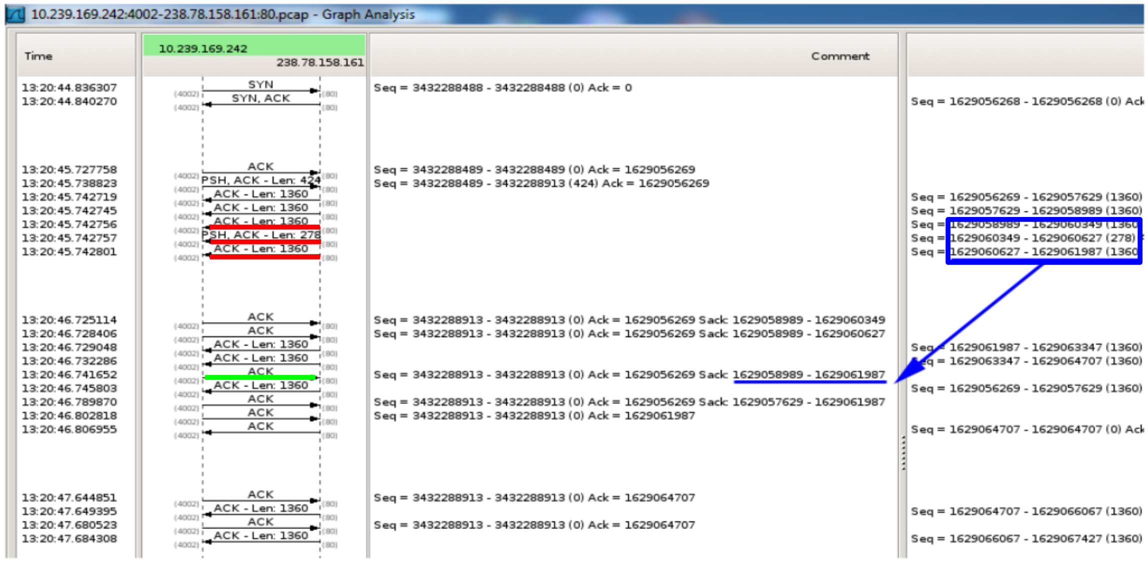

Figure 6.2: Extended Flow Graph in Wireshark

Dividing the 'Comments' column to two columns based on which direction the TCP-PDUs flow: The 'Comments' column is divided into two columns. The left column displays the information of TCP-PDUs represented by right-pointing arrows, and the right column displays the information of TCP-PDUs represented by left-pointing arrows.

Displaying TCP SACK information: TCP SACK information is displayed. Also, when an ack (which contains SACKs) is clicked, those TCP-PDUs which are SACKed by the clicked ack are highlighted. In Figure 6.2, when the ack which contains SACK 1629058989 - 1629061987 is clicked (highlighted in green color), three TCP-PDUs are highlighted in red color.

Displaying space between TCP-PDUs to emphasize gaps in time: Based on the time intervals between TCP-PDUs, vertical spaces are added between arrows.

These updates to wireshark made it significantly easier to analyze a TCP flow and decide which case holds for a candidate reneging instance [18]. This extension can be downloaded at: http://www.cis.udel.edu/~amer/PEL/Wireshark_TCP_ flowgraph_patch.tar

Chapter 7

SUMMARY AND CONCLUSIONS

This dissertation investigated two issues related to the transport layer and proposed solutions to address these issues. Each chapter ended with a discussion of corresponding future work. This chapter summarizes our contributions for each issue, and concludes the dissertation.

7.1 Issue I: Reneging and NR-SACKs

TCP is designed to tolerate reneging. This design has been challenged since (i) reneging rarely occurs in practice, and (ii) even when reneging does occur, it alone generally does not help the operating system resume normal operation when the system is starving for memory. In current MPTCP standard, an MPTCP receiver cannot report the reception of out-of-order data to an MPTCP sender. We investigated how freeing received out-of-order PDUs from the send buffer can improve end-to-end performance when send buffer blocking occurs in both TCP and MPTCP. We introduced and implemented NR-SACKs for both TCP and MPTCP in the Linux kernel.

Preliminary result for TCP NR-SACKs showed that (i) TCP data transfers with NR-SACKs never perform worse than those without NR-SACKs, and (ii) NR-SACKs can improve end-to-end throughput when send buffer blocking occurs. We are currently doing a collaboration study between UD and ISAE-SUPAERO of quantifying potential gains of TCP NR-SACKs over an actual long delay, lossy satellite link in CNES.

Preliminary result for MPTCP NR-SACKs showed that (i) MPTCP data transfers with NR-SACKs never perform worse than those without NR-SACKs, and (ii) NR-SACKs can improve end-to-end throughput in MPTCP when send buffer blocking occurs.

7.2 Issue II: MPTCP Scheduling

An important component of MPTCP is the scheduler. Whenever an MPTCP sender wants to send data, the scheduler needs to decide on which subflow to send each byte. During experiments on MPTCP NR-SACKs, we found a problem of the default scheduler of the Linux MPTCP. We investigated two different scheduling policies for MPTCP, and addressed these two scheduling policies to improve application performance.

We explained problems with the default scheduler used by Linux MPTCP, and proposed the design of a scheduler which based on not only a subflow's 'speed' but also the subflow's congestion. Preliminary empirical result showed that our proposed scheduler improves the throughput in MPTCP by alleviating the problems caused by the default scheduler.

We also used one-way communication delay of a TCP connection to design an MPTCP scheduler that transmits data out-of-order over multiple paths such that their arrival is in-order. Our Linux implementation showed our proposed scheduler can reduce receive buffer utilization, and increase throughput when a small receive buffer size results in receive buffer blocking.

BIBLIOGRAPHY

[1] A. Ford, C. Raiciu, M. Handley, and O. Bonaventure, *TCP Extensions for Multipath Operation with Multiple Addresses*, draft-ietf-mptcp-multiaddressed-09. IETF Internet draft, June 2012.

[2] M. Scharf, A. Ford, *MPTCP Application Interface Considerations*, draft-ietf-mptcp-api-07, IETF Internet draft, January 2013.

[3] C. Raiciu, M. Handley, D. Wischik, *Coupled Congestion Control for Multipath Transport Protocols*, RFC 6356, October 2011.

[4] D. Wischik, C. Raiciu, A. Greenhalgh, and M. Handley, *Design, Implementation and Evaluation of Congestion Control for Multipath TCP*. 8th USENIX Symposium on Networked Systems Design and Implementation, Boston, Massachusetts, USA, March 2011.

[5] S. Barré, C. Paasch, and O. Bonaventure, *MPTCP: From Theory to Practice*. Networking 2011, Valencia, Spain, May 2011.

[6] O. Bonaventure, M. Handley and C. Raiciu, *An overview of Multipath TCP*. USENIX login;, October 2012.

[7] S. Barré, *Implementation and assessment of Modern Host-based Multipath Solutions*. PhD Dissertation, Universit catholique de Louvain, 2011.

[8] A. Kostopoulos, H. Warma, T. Leva, B. Heinrich, A. Ford and L. Eggert, *Towards Multipath TCP Adoption: Challenges and Opportunities*, 6th EURO-NF Conference on Next Generation Internet (NGI), 2010.

[9] M. Mathis and J. Mahdavi, *Forward Acknowledgement: Refining TCP Congestion Control*. ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, New York, NY, USA, 1996.

[10] J. Postel, *Discard Protocol*. RFC 863, May 1983.

[11] P. Amer, E. Lochin, F. Yang and S. Trang, *TCP with Non-Renegable SACKs over Satellite Links*, UD-ISAE Collaboration Study (in progress).

[12] J. Iyengar, P. Amer, and R. Stewart, *Concurrent Multipath Transfer Using SCTP Multihoming Over Independent End-to-end Paths*. IEEE/ACM Trans on Networking, 14(5), October 2006.

[13] P. Amer, M. Becke, T. Dreibholz, N. Ekiz, J. Iyengar, P. Natarajan, R. Stewart, and M. Tüxen, *Load Sharing for the Stream Control Transmission Protocol (SCTP)*, draft-tuexen-tsvwg-sctp-multipath-05. IETF Internet draft, September 2012.

[14] N. Ekiz, P. Amer, P. Natarajan, R. Stewart and J. Iyengar, *SCTP Data Acknowledgement with Non-renegable Selective Acks (NR-SACKs)*, draft-natarajan-tsvwg-sctp-nrsack. IETF Internet draft, February 2011.

[15] P. Natarajan, N. Ekiz, E. Yilmaz, P. Amer, J. Iyengar, and R. Stewart, *Non-Renegable Selective Acks (NR-SACKs) for SCTP*. IEEE International Conference on Network Protocols, Orlando, Florida, USA, October 2008.

[16] N. Ekiz and P. Amer, *Transport Layer Reneging* (submitted for publication).

[17] P. Natarajan, *Leveraging Transport Services for Improved Application Performance*. PhD Dissertation, CIS Department, University of Delaware, 2009.

[18] N. Ekiz, *Transport Layer Reneging*. PhD Dissertation, CIS Department, University of Delaware, May 2012.

[19] E. Yilmaz, N. Ekiz, P. Amer, J. Leighton, F. Baker, and R. Stewart, *Throughput Analysis of Non-Renegable Selective Acknowledgments (NR-SACKs) for SCTP*. Computer Communications, 33(16):1982–1991, October 2010.

[20] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow, *TCP Selective Acknowledgment Options*. RFC 2018, October 1996.

[21] H. Adhari, T. Dreibholz, M. Becke, E.P. Rathgeb and M. Tüxen, *Evaluation of Concurrent Multipath Transfer over Dissimilar Paths*. 1st International Workshop on Protocols and Applications with Multi-Homing Support, Singapore, 2011.

[22] F. Yang and P. Amer, *Non-renegable Selective Acks (NR-SACKs) for MPTCP*. The 3rd International Workshop on Protocols and Applications with Multi-Homing Support, Barcelona, Spain, March, 2013.

[23] N. Ekiz, P. Amer and F. Yang, *Causing remote hosts to reneg*. The 7th Workshop on Performance Modeling and Evaluation in Computer and Telecommunication Networks, Nassau, Bahamas, July 2013.

[24] C. Cetinkaya and E. Knightly, *Opportunistic traffic scheduling over multiple network paths*. 23rd AnnualJoint Conference of the IEEE Computer and Communications Societies, INFOCOM 2004.

[25] C. Cetinkaya, *Improving the efficiency of multipath traffic via opportunistic traffic scheduling*. Computer Networks, 2007.

[26] A. Singh, C. Goerg, A. Timm-Giel, M. Scharf and T.R. Banniza, *Performance Comparison of Scheduling Algorithms for Multipath Transfer.* .

[27] D. Sarkar, *A concurrent multipath TCP and its markov model*, IEEE International Conference on Communications (ICC'06), 2006.

[28] Y. Dong, D. Wang, N. Pissinou and J. Wang, *Multi-path load balancing in transport layer*, 3rd EuroNGI Conference on Next Generation Internet Networks, 2007.

[29] P. Key, L. Massoulié and D. Towsley, *Path selection and multipath congestion control*, 26th IEEE International Conference on Computer Communications, INFOCOM 2007.

[30] W. Zhang, Q. Wu, W. Yang and H. Li, *Reliable Multipath Transfer Scheduling Algorithm Research and Prototype Implementation.* 34th Proceedings of the Asia Pacific Advanced Network, Colombo, Sri Lanka, August 2012.

[31] B. Wang, W. Wei, J. Kurose, D. Towsley, K.R. Pattipati, Z. Guo and Z. Peng, *Application-layer multipath data transfer via TCP: schemes and performance tradeoffs.* Performance Evaluation, 2007.

[32] F. Yang, P. Amer and N. Ekiz, *A Scheduler for MPTCP.* 22nd International Conference on Computer Communications and Networks, Nassau, Bahamas, July 2013.

[33] F. Yang, Q. Wang and P. Amer, *Out-of-order Transmission for In-order Arrival Scheduling for MPTCP.* 4th International Workshop on Protocols and Applications with Multi-Homing Support, Victoria, Canada, May, 2014.

[34] F. Yang and P. Amer, *Using One-way Communication Delay for In-order Arrival MPTCP Scheduling.* 9th EAI ChinaCom 2014, Maoming, China, August, 2014.

[35] *MultiPath TCP - Linux Kernel Implementation.* http://mptcp.info.ucl.ac.be/.

[36] S. Nguyen, X. Zhang, T. Nguyen and G. Pujolle, *Evaluation of Throughput Optimization and Load Sharing of Multipath TCP in Heterogeneous Networks.* WOCN 2011, New Orleans, Louisiana, 2011.

[37] A.S. Carmelita Görg, A. Timm-Giel, and M. Thomas-Ralf Banniza, *Performance Evaluation of Multipath TCP Linux Implementations.* Wrzburg Workshop on IP: Joint ITG and Euro-NF Workshop Visions of Future Generation Networks. 2011.

[38] H. Adhari, T. Dreibholz, M. Becke, E.P. Rathgeb and M. Tüxen, *Evaluation of Concurrent Multipath Transfer over Dissimilar Paths.* 1st International Workshop on Protocols and Applications with Multi-Homing Support, Singapore, 2011.

[39] H. Hsieh and R. Sivakumar, *pTCP: An End-to-end Transport Layer Protocol for Striped Connections*. IEEE International Conference on Network Protocols, Paris, France, November 2002.

[40] K. Rojviboonchai, T. Osuga, and H. Aida, *R-M/TCP: Protocol for Reliable Multipath Transport Over the Internet*. AINA 2005, Taiwan, 2005.

[41] M. Zhang, J. Lai, and A. Krishnamurthy, *A Transport Layer Approach for Improving End-to-end Performance and Robustness Using Redundant Paths*. 2004 USENIX Annual Technical Conference, Boston, MA, USA.

[42] L. Xu, K. Harfoush, and I. Rhee, *Binary Increase Congestion Control for Fast, Long Distance Networks*. 23th AnnualJoint Conference of the IEEE Computer and Communications Societies, Hong Kong, China, March, 2004.

[43] L.S. Brakmo, and L.L. Peterson, *TCP Vegas: End to End Congestion Avoidance on a Global Internet*. IEEE Journal on Selected Areas in Communications, October 1995.

[44] C. Jin, D.X. Wei, S.H. Low, *FAST TCP: Motivation, Architecture, Algorithms, Performance*. 23rd AnnualJoint Conference of the IEEE Computer and Communications Societies, Hong Kong, China, March, 2004.

[45] R. Jain, *A Delay-based approach for Congestion Avoidance in Interconnected heterogeneous computer networks*. Comput. Commun. Rev. October, 1989.

[46] J. Martin, A. Nilsson, and I. Rhee. *Delay-Based Congestion Avoidance for TCP*. IEEE/ACM Transactions on Networking, June 2003.

[47] R. Khalili, N. Gast, M. Popovic, U. Upadhyay, J.Y. Le Boudec, *Non Pareto-Optimality of MPTCP: Performance Issues and a Possible Solution*, ACM CoNEXT, 11/2012.

[48] J. Iyengar, P. Amer and R. Stewart, *Receive Buffer Blocking in Concurrent Multipath Transfer*. IEEE Global Telecommunications Conference, 2005.

[49] M. Li, A. Lukyanenko and Y. Cui, *Network Coding Based Multipath TCP*, IEEE Conference on omputer Communications Workshops, 2012.

[50] *NorNet Project*. http://www.nornet-testbed.no/.

[51] *Linux Advanced Routing and Traffic Control*. http://www.lartc.org/.

[52] Q. Pan, X. Luo, H. Xiao, *An Approach to Improve the Accuracy of One-Way Delay Measurements*, Communications in Computer and Information Science, 2011

[53] A. Hernandez, E. Magana, *One-way Delay Measurement and Characterization*, ICNS 2007, Pamplona, Spain, 08/2007

[54] J. Hee, C. Yoo, *One-way Delay Estimation and its Application*, Computer Communications, 2005

[55] M. Belshe, R. Peon, *SPDY Protocol*, http://mbelshe.github.com/SPDY-Specification/draft-mbelshe-spdy-00.xml

[56] Zlib: A Massively Spiffy Yet Delicately Unobtrusive Compression Library, http://www.zlib.net/

[57] The 1000 Most-Visited Sites on the Web by Google, http://www.google.com/adplanner/static/top1000/

[58] Training Data: HTTP Request Headers, http://www.cis.udel.edu/~amer/PEL/SPDY/HTTP_requests_training

[59] Training Data: HTTP Reply Headers, http://www.cis.udel.edu/~amer/PEL/SPDY/HTTP_replies_training

[60] Evaluation Data: HTTP Request Headers, http://www.cis.udel.edu/~amer/PEL/SPDY/HTTP_requests_evaluation

[61] Evaluation Data: HTTP Reply Headers, http://www.cis.udel.edu/~amer/PEL/SPDY/HTTP_replies_evaluation

Appendix A

PACKET FORMATS OF NON-RENEGABLE SELECTIVE ACKNOWLEDGMENTS (NR-SACKS) FOR MPTCP

A.1 Modified Multipath Capable (MP_CAPABLE) Option

Before sending/receiving NR-SACKs, two end hosts must negotiate NR-SACK usage during the connection initiation phase. A proposed modified MP_CAPABLE option is shown in Figure A.1. Two bits — 'N' and 'n' — are used. During the three-way handshake, N bits of the two SYNs (SYN and SYN/ACK) indicates "NR-SACK capability of the SYN's sender". The decision of using NR-SACKs in data transfer is confirmed by the setting of N bit in the third packet (the ACK). N bit in the ACK packet = $N_{SYN} AND N_{SYN/ACK}$, which means NR-SACK is used only if both endpoints are NR-SACK capable.

In a packet, the n bit has meaning only if $N = 1$, otherwise the n bit MUST be ignored. $n = 1$ indicates the size of one NR-SACK block is 6 bytes, and $n = 0$ means the size of one NR-SACK block is 8 bytes. The reason for using variant NR-SACK block size is explained in section 3.2. The decision of the size of one NR-SACK block in data transfer is confirmed by the setting of n bit in the ACK packet. n bit in the ACK = $n_{SYN} AND n_{SYN/ACK}$, which means the size of one NR-SACK block is 6 bytes only if both endpoints set $n = 1$ in their SYNs, else the size is 8 bytes.

A.2 Modified Data Sequence Signal (DSS) Option including NR-SACK

Before talking about the proposed DSS option, consider how many NR-SACK blocks can be present in the TCP option field. During unidirectional MPTCP data transfers, the NR-SACKs are carried by pure acks (acks without application data). The maximum size of the TCP option field is 40 bytes. A timestamp option occupies 12

```
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+---------------+---------------+-------+-------+-+-+-+-------+-+
|     Kind      |    Length     |Subtype|Version|C|N|  (rsvd) |S|
+---------------+---------------+-------+-------+-+-+-+-------+-+
|                   Option Sender's Key (64 bits)               |
|                                                               |
|                                                               |
+---------------------------------------------------------------+
|                  Option Receiver's Key (64 bits)              |
|                     (if option Length == 20)                  |
|                                                               |
+---------------------------------------------------------------+
```

Figure A.1: Modified MP_CAPABLE Option

bytes (with padding) leaving 28 bytes. Assuming no SACK information, a DATA ACK needs 8 or 12 bytes (depending on flag 'a'), thus only up to 20 bytes can be used for NR-SACKs. To decrease the number of bytes needed to represent one NR-SACK block, the left and right edge values of a reported NR-SACK block can be defined relative to the DATA ACK value. For example, if the MPTCP receiver receives out-of-order data with DSNs from DSN_{start} to DSN_{end}, the left and right edge values of the reported NR-SACK block are $DSN_{start} - DATAACK$ and $DSN_{end} + 1 - DATAACK$, respectively. With 6-byte NR-SACK block, up to 3 blocks can be present and out-of-order bytes within 2^{24} (16MB) from the DATA ACK can be reported. When an MPTCP receive buffer size is <= 16MB, 6 bytes is sufficient. However, when an MPTCP receive buffer size is >= 16MB, 6 bytes may not be enough. In this situation, the size of one NR-SACK block can be negotiated to be 8 bytes during connection establishment. Only 2 NR-SACK blocks will fit if the size of one NR-SACK block is 8 bytes.

The proposed modified DSS options with NR-SACKs are shown in Figure A.2 (each NR-SACK is 6 bytes) and A.3 (each NR-SACK is 8 bytes). A 2-bit unsigned integer — 'C' — is used to indicate the number of presented NR-SACK blocks. When the size of one NR-SACK block is 6 bytes and 1 or 3 NR-SACK blocks are present, two bytes paddings are used for alignment. The NR-SACKs can be present only when DATA ACK is present, and NR-SACKs yield the TCP option space to all TCP and

```
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4  5  6 7 8 9 0 1
+---------------+---------------+-------+-----------------------+
|     Kind      |    Length     |Subtype|(reserved)|  C  |F|m|M|a|A|
+---------------+---------------+-------+-----------------------+
|          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)      |
+-------------------------------+---------------+---------------+
|        Left Edge of NR-SACK Block #1          |  Right Edge of
+-------------------------------+---------------+---------------+
  NR-SACK Block #1              |         Left Edge of NR-SACK
+---------------+---------------+-------------------------------+
  Block #2      |       Right Edge of NR-SACK Block #2          |
+---------------+-------------------------------+---------------+
|        Left Edge of NR-SACK Block #3          |  Right Edge of
+-------------------------------+---------------+---------------+
  NR-SACK Block #3              |            Padding            |
+-------------------------------+-------------------------------+
```

Figure A.2: Modified DSS Option (each NR-SACK is 6 bytes)

other MPTCP options. As specified for SACKs in TCP, NR-SACKs always report the block containing the most recently received data, because this approach provides a MPTCP sender with the most up-to-date information about the state of a MPTCP receive buffer.

```
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4   5  6 7 8 9 0 1
+---------------+---------------+-------+-------------------------+
|     Kind      |    Length     |Subtype|(reserved)|  C  |F|m|M|a|A|
+---------------+---------------+-------+-------------------------+
|           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)        |
+-------------------------------+---------------------------------+
|          Left Edge of NR-SACK Block #1 (4 octets)               |
+-----------------------------------------------------------------+
|          Right Edge of NR-SACK Block #1 (4 octets)              |
+-----------------------------------------------------------------+
|          Left Edge of NR-SACK Block #2 (4 octets)               |
+-----------------------------------------------------------------+
|          Right Edge of NR-SACK Block #2 (4 octets)              |
+-----------------------------------------------------------------+
```

Figure A.3: Modified DSS Option (each NR-SACK is 8 bytes)