5th Federated and Fractionated Satellite Systems Workshop
November 2–3, 2017, ISAE SUPAERO – Toulouse, France

TCP Non-Renegable Selective Acknowledgments (NR-SACKs) and benefits for space and satellite communications

Fan Yang*, Paul D. Amer*, Si Quoc Viet Trang†, and Emmanuel Lochin†

*CIS Department of University of Delaware, Newark, DE 19716, USA ,
firstname.name@udel.edu
†Université de Toulouse ISAE-SUPAERO, DISC-RESCOM, Toulouse, France ,
firstname.name@isae-supaero.fr

Abstract

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. 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 TCP. 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% and particularly over long-delay links such as GEO satellite links. The tradeoff for this potential gain is a change to the semantics of the TCP send buffer requiring the more complex management of non-contiguous PDUs. We investigate potential application performance gains when TCP receiver implements NR-SACKs and present empirical results on a real satellite link in the Centre National d'Études Spatiales (CNES) France's agency responsible for shaping and implementing its space policy in Europe.

Keywords

Satellite communications; TCP; TCP Reneging; Selective Acknowledgment

1 Introduction

Reliable transport protocols (such as TCP and SCTP) employ two data acknowledgment mechanisms: (i) cumulative acknowledgments (CUMACK) indicate data that has been received in-sequence, and (ii) selective acknowledgments (SACKs) indicate the data that has been received, whether it is out-of-order or not. While cumulative acknowledged data becomes solely a receiver's responsibility, SACKed data does not. SACKed out-of-order data is implicitly renegable; that is, a receiver may SACK data and later discard it [1]. The reneging feature induces that a transport sender must maintain copies of SACKed data in the send buffer until they are cumulatively acknowledged which can result in a non-negligible utilization of the sending buffer and sending buffer blocking.

TCP's design to tolerate reneging has been challenged since (i) reneging rarely occurs in practice [2], 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 [2]. If a TCP receiver never renegs, SACKed data is wastefully stored in the send buffer until cumulatively acknowledged.

Non-Renegable Selective Acknowledgments (NR-SACKs) were introduced in [3] to improve end-to-end throughput performance when sender buffer blocking occurs. 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 CUMACK. NR-SACKs have been evaluated for SCTP, SCTP with Concurrent Multipath Transmission (CMT) and multipath TCP (MPTCP), and results show NR-SACKs not only reduce sender's memory requirements, but also improve the end-to-end throughput under certain conditions [4, 5, 6, 7]. Quick UDP Internet Connections (QUIC) [8], generally considered as HTTP2 + TLS + TCP and designed by Google, already supports a functionality similar to NR-SACKs. In a QUIC ACK frame, a QUIC receiver reports the largest observed packet number, and up to 256 NACK ranges which represent packet numbers which are considered to be lost. After receiving an acknowledgment, the QUIC sender can free those packets (with sequence numbers less than the highest received sequence number) which are not reported in NACKs from the send buffer.

In this work, we investigate potential application performance gains if a TCP receiver is designed never to reneg, and likewise uses NR-SACKs This paper is organized as follows. Section II explains potential performance gains by prohibiting reneging in TCP. Section III briefly explains our implementation of TCP NR-SACKs in Linux kernel. Section IV analyzes empirical results for TCP data transfers with NR-SACKs vs. without NR-SACKs on a small testbed topology in our lab. Section V and present empirical results of NR-SACKs on a real satellite link in the Centre National d'Études Spatiales (CNES) France's agency responsible for shaping and implementing its space policy in Europe. Section VI concludes our work.

2 Potential Performance Gains by Prohibiting Reneging in TCP

Figure 1 illustrates how the reneged TCP transfer can result in sending buffer blocking (1), and how a non-regenable approach would result in better utilization of the available resource (2). The TCP sender is not blocked 'wastefully' maintaining copies of SACKed data. Instead the send buffer has room for transmitting new application data.

To gain insight to the performance penalty incurred by TCP tolerating reneging, consider the example in Figure 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 CUMACK come back to the sender, the cumulatively acknowledged 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 cumulatively acknowledged 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 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 cumulatively acknowledged. This situation is referred to as *send buffer blocking*.

From Figures 1 and 2, we have following observations:

First, the data transmission is blocked due to loss in Figure 1. For a TCP connection with a lossy and long-delay link, send buffer blocking can seriously decrease the throughput. NR-SACKs alleviate send buffer blocking hence higher throughput is achieved in Figure 2.

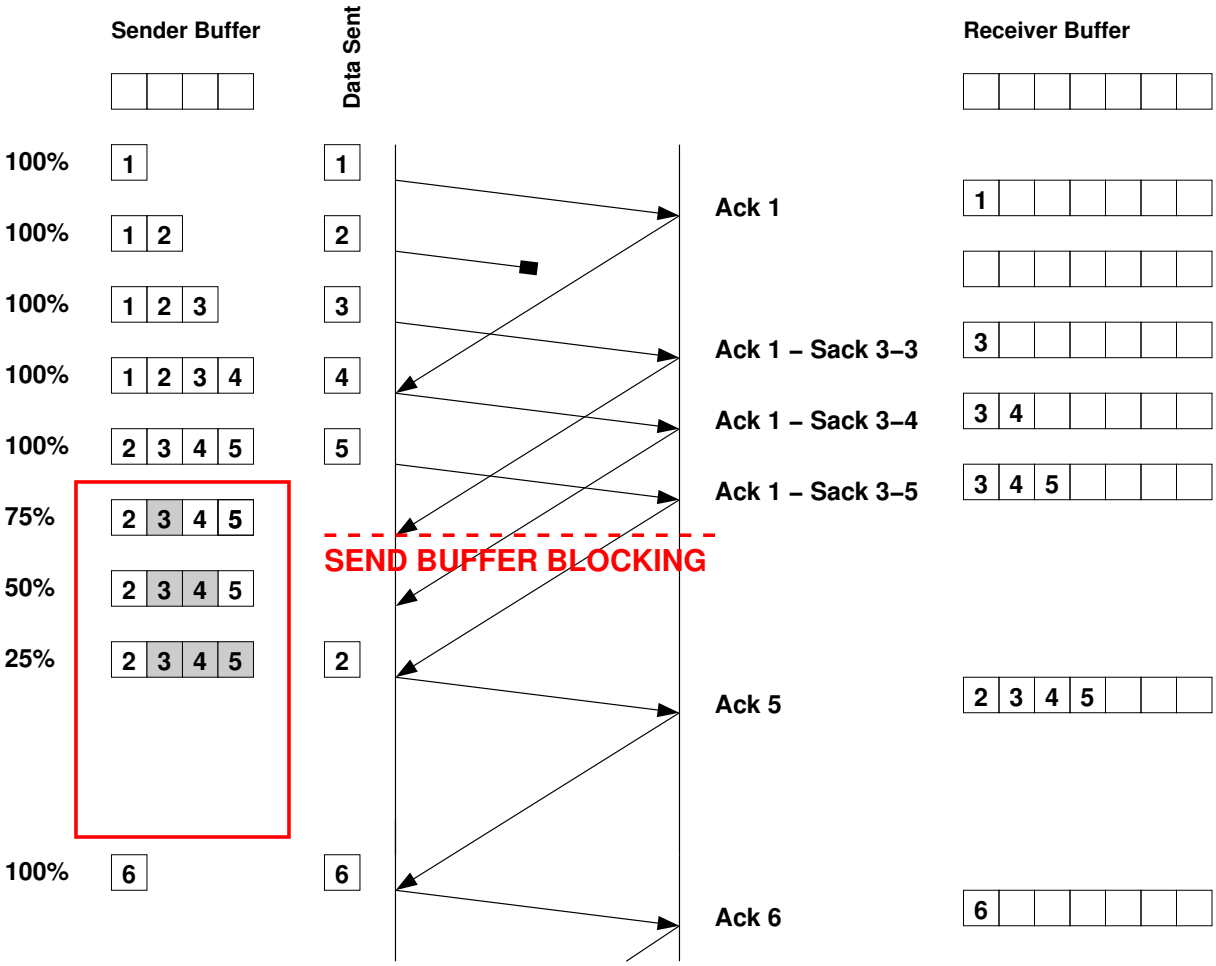

Figure 1: TCP data transfer w/o NR-SACK (TCP vanilla)

Second, 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.

Third, 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 circumstances, there is no advantage to having a receive window larger than the send window (as is demonstrated in Figure 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. However, with this change, the send buffer may have gaps. For example, in Figure 2, after TCP-PDU 3 is freed by NR-SACK 3-3, the send buffer is no longer contiguous. With this change in the send window semantics, 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.)

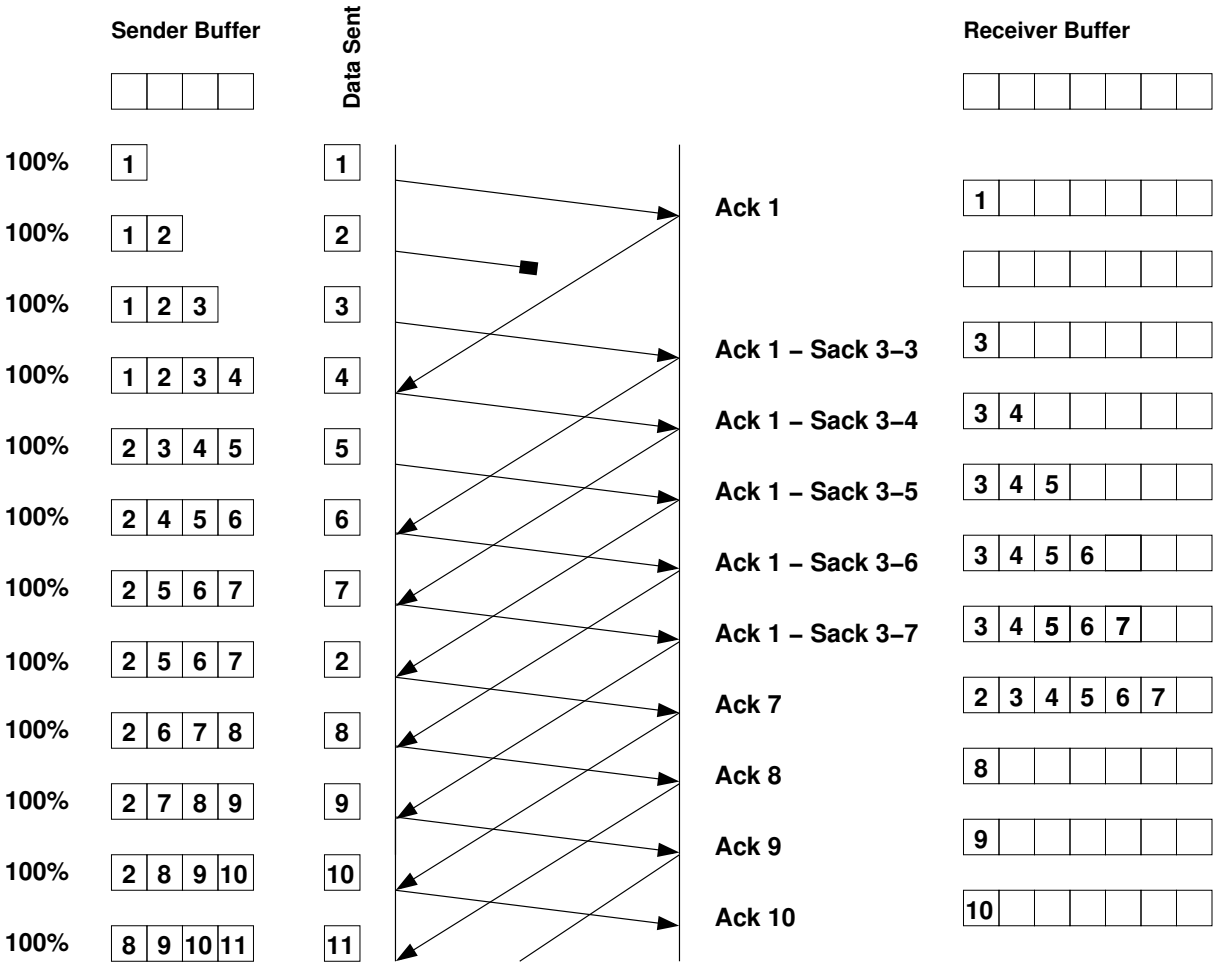

Figure 2: TCP data transfer with NR-SACK

3 Implementation

In Linux, the *sk_buff* structure (skb) represents data that is about to be transmitted by a sender or has been received by a receiver. Each skb in the TCP retransmission queue is tagged by a *sacked* field indicating the current state of the skb (e.g., SACKed, lost, retransmitted, etc). 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 the congestion control and flow control mechanisms. If a TCP sender simply frees SACKed skbs, the sender will have wrong estimates of the current state of the network, and these wrong estimates will mislead the congestion control mechanism. Therefore, NR-SACKs can only free data sections of a SACKed skb, and must maintain the control information sections to allow a TCP sender have correct estimates of the current network state. The complete implementation details of TCP NR-SACKs in the Linux kernel can be found in [9] and the kernel patch is given in Appendix for information purpose.

4 Experimental Design I

The testbed (Figure 3) of experiment I is composed of a Cisco Linksys E1200 router and three computers: two TCP senders and one TCP receiver. 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 the normal TCP sender and the receiver, and the other between the

NR-SACK TCP sender and the receiver. 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.

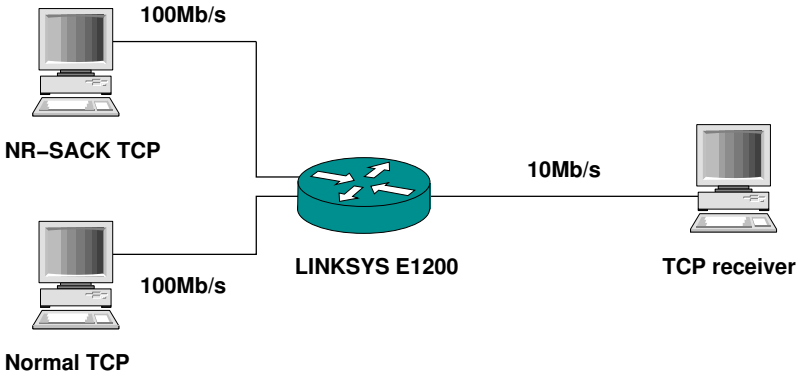

Figure 3: Testbed topology I

4.1 Experimental Parameters

In Linux, the default upper limit of the TCP send buffer size on the TCP senders is 905KB. The performance of NR-SACKs is tested under six different send buffer sizes {22KB, 44KB, 90KB, 181KB, 362KB, 905KB}, three different loss rates {0%, 1%, 5%} and three different one way 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 [10].

4.2 Results

To evaluate the performance of TCP data transfers with NR-SACKs vs. without NR-SACKs, we employ the metric *throughput gain* defined in [6] 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. When we analyze how throughput gain varies as the loss rate and one-way delay change, we use a *region of gain* metric [6] defined as the send buffer size interval, [a, b], where any send buffer size from a to b inclusive results in an expected throughput gain of at least 5%.

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 [5] and under certain parameter configurations, NR-SACKs would improve the end-to-end throughput. Figure 4 shows the throughput gains for all parameter combinations tested.

With no loss, the number of runs was one or two because results were identical. With random loss, 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, 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 have not arrived at the receiver. 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 $\overline{\mathrm{cwnd}}$):

$$\mathrm{RtxQ\ size} \leq \min\left(\mathrm{BDP}, \overline{\mathrm{cwnd}}\right) \tag{1}$$

For a given delay, increased loss results in a smaller $\overline{\mathrm{cwnd}}$. For a given loss rate, longer delay results in a larger BDP. In the remaining of this section, we discuss on the impact of loss rate and delay on throughput gain of NR-SACKs.

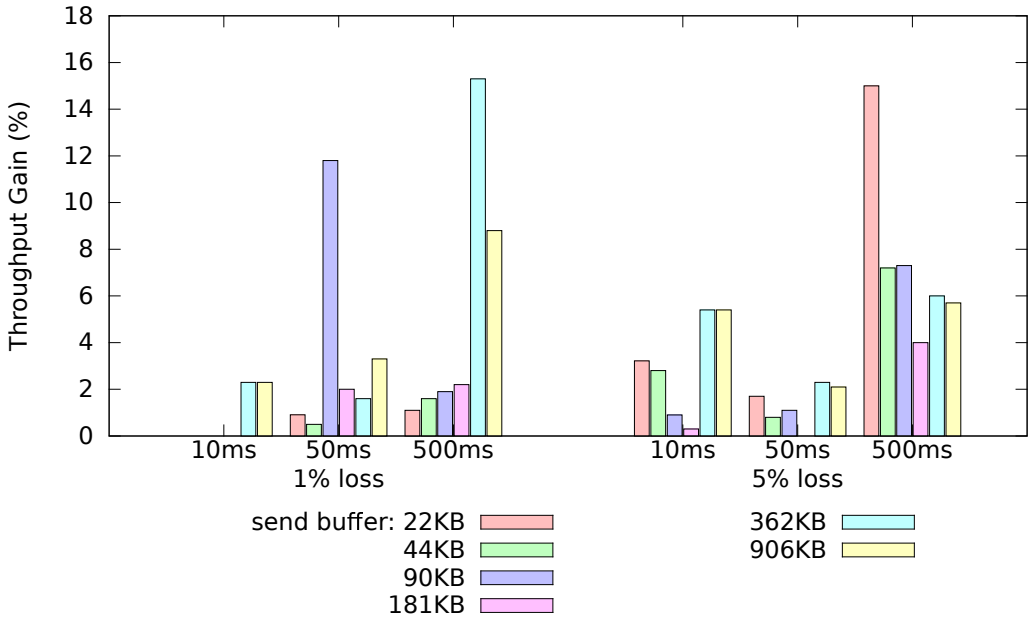

Figure 4: NR-SACKs throughput gain with various send buffer sizes

We observed regions of gain for both loss rates. The region of gain for 1% loss was [212KB, 905KB], and for 5% loss was [10KB, 155KB]. As the loss rate increases, cwnd decreases and hence the send buffer blocking region becomes smaller, thus indicating less of an impact of NR-SACKs.

We did not observe any obvious region of gain with 10ms delay, and the regions of gain with 50ms and 500ms delays were [65KB, 160KB] and [212KB, 905KB], respectively. As stated in section 4.2, a longer delay results in a larger BDP. As delay increases, BDP increases and hence the send buffer blocking region becomes larger. In this case, NR-SACKs provide potential throughput gains.

5 Experiment Design II

Encouraged by the positive results in our lab (not presented in this extended abstract), a collaboration study began between University of Delaware (UD) and Institut Supérieur de l'Aéronautique et de l'Espace (ISAE-SUPAERO) to empirically quantify potential gains of TCP NR-SACKs over a long delay, lossy satellite link at France's Centre National d'Études Spatiales (CNES)[1].

Figure 5 shows the testbed. Three computers (one normal TCP sender, one NR-SACK sender and one TCP receiver) are physically located at CNES. The TCP senders and receiver are connected via a real satellite link. The RTT and bandwidth of the satellite link is approximately 610ms and 80Kbps, respectively. The traffic is generated by transferring a 1MB file from a TCP sender to the receiver. The performance of TCP NR-SACKs vs TCP normal is tested under four different send buffer sizes {22KB, 44KB, 90KB, 181KB} and five different loss rates {0%, 0.5%, 1%, 2%, 5%}.

Figure 6 shows the average throughput gains with NR-SACKs. Each data point represents the average throughput gain of 50 transfers with and 50 transfers without NR-SACKs. These results confirm our hypothesis that TCP data transfers with NR-SACKs always perform as well as those without NR-SACKs. We did not observe obvious region of gain with 2% and 5% losses. The right edges of throughput gains with loss rates 0%, 0.5% and 1% are 44KB, 38KB and 34KB, respectively. This result is consistent with those presented in the previous section: that the region of gain moves left towards smaller buffer sizes as the loss rate increases. We also observed that, within the region of gain, the maximum throughput gain of a higher loss rate would be greater than that of a smaller loss rate. When the loss rate is 1%, the throughput gain can reach as high as 6.8% under a 22KB send buffer.

[1] http://www.cnes.fr/

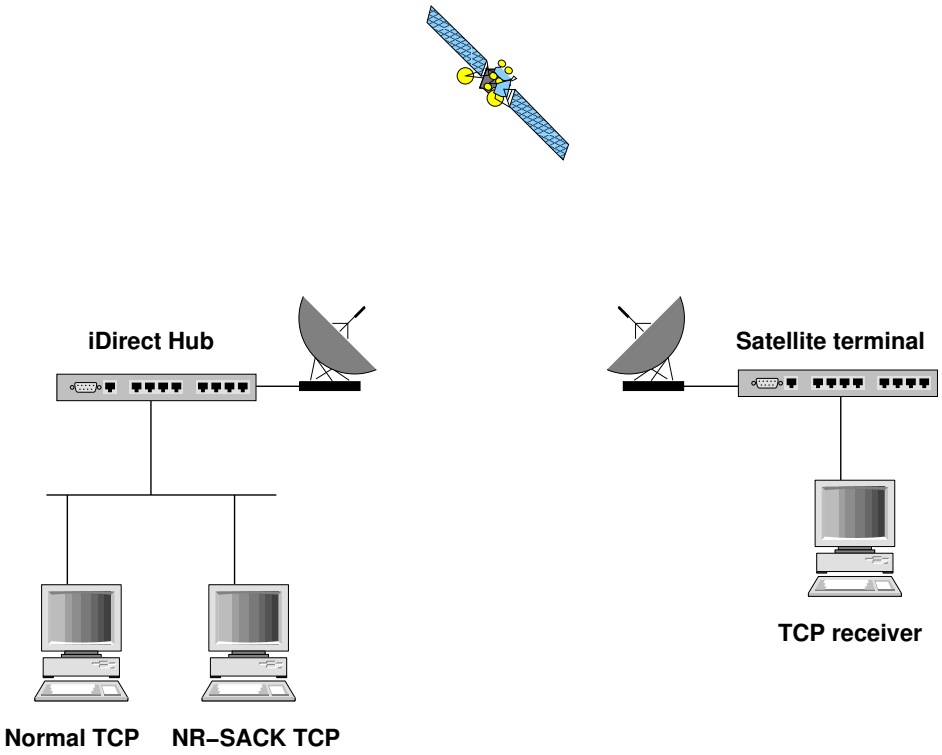

Figure 5: Topology for testing TCP NR-SACKs at CNES

6 Discussion

Because the data receiver is allowed to discard SACKed data, when a retransmit timeout occurs the data sender **MUST** ignore prior SACK information in determining which data to retransmit [1]. This **MUST** should have been a **SHOULD**. But the point of SACKs is to make TCP robust, efficiency is not the goal. Currently, middleboxes could do incorrect things. For example, a NAT rewrites a connection which has got symbolic IP addresses in it and rewrites IP addresses, then the offset of the segment is different. Also devices can randomize sequence numbers but don't fix SACK blocks, so SACK blocks are out of the window. So our research is trying to demonstrate freeing received data from the send buffer is beneficial and new transport protocol (e.g., QUIC) designs can adopt this mechanism rather than add NR-SACKs to the TCP standard.

7 Conclusion

In this work, we introduced NR-SACKs to TCP and investigated their impact in situations where a TCP receiver never renegs. We extended the Linux TCP to support NR-SACKs. Based on both experiments, we concluded 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 in TCP when send buffer blocking occurs. The positive gains demonstrated in our experiments were mostly for small buffer sizes not likely to be used in practical situations. However, if the satellite link were higher bandwidth, thus increasing the BDP, we expect gains for larger buffer sizes as buffer blocking is more likely to occur.

Acknowledgements

The authors would like to thank Nicolas Kuhn for useful comments and CNES for providing us the satellite testbed CESARS.

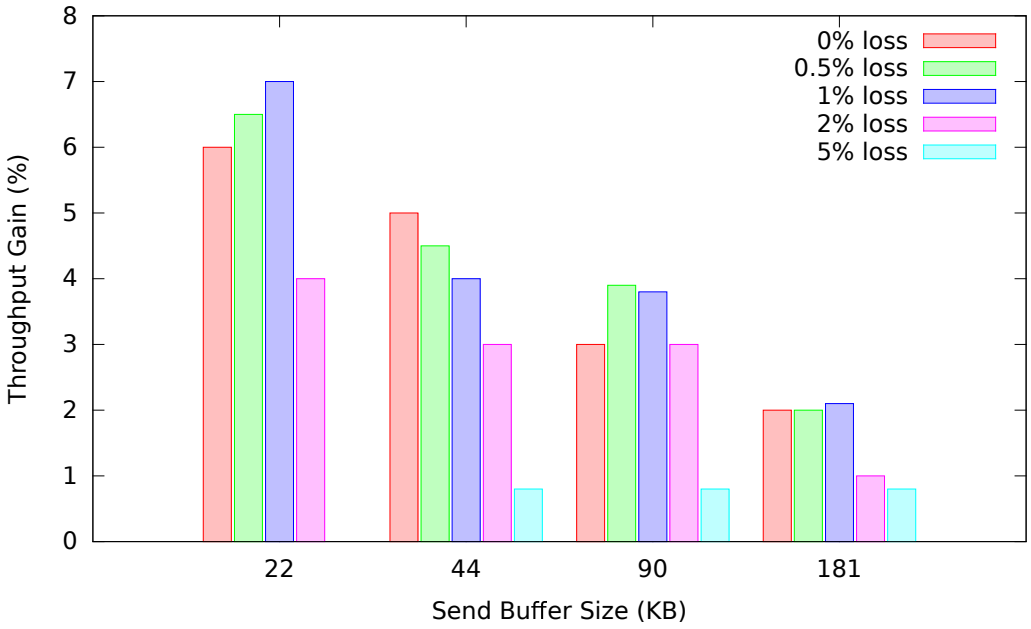

Figure 6: Throughput gain with NR-SACKs over satellite link

References

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

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

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

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] R. Hamilton, J. Iyengar, I. Swett and A. Wilk, *QUIC Wire Layout Specification*. `http://bit.ly/1s2RHWJ`.

[9] F. Yang, *Non-Renegable Selective Acknowledgments and Scheduling for TCP and Multipath TCP*. PhD Dissertation, CIS Department, University of Delaware, 2015.

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

Appendix

```
diff -rcNP linux-3.0.1-vanilla/include/linux/skbuff.h linux-3.0.1-tcp-nrsacks-dev/include/linux/skbuff.h
*** linux-3.0.1-vanilla/include/linux/skbuff.h 2011-08-05 00:59:21.000000000 -0400
--- linux-3.0.1-tcp-nrsacks-dev/include/linux/skbuff.h 2014-04-20 15:58:59.000000000 -0400
***************
*** 486,492 ****
  {
   return (struct rtable *)skb_dst(skb);
  }
!
  extern void kfree_skb(struct sk_buff *skb);
  extern void consume_skb(struct sk_buff *skb);
  extern void        __kfree_skb(struct sk_buff *skb);
--- 486,493 ----
  {
   return (struct rtable *)skb_dst(skb);
  }
! /* UD_PEL TCP-NRSACKs */
! extern int skb_free_frags(struct sk_buff *skb);
  extern void kfree_skb(struct sk_buff *skb);
  extern void consume_skb(struct sk_buff *skb);
  extern void        __kfree_skb(struct sk_buff *skb);
diff -rcNP linux-3.0.1-vanilla/net/core/skbuff.c linux-3.0.1-tcp-nrsacks-dev/net/core/skbuff.c
*** linux-3.0.1-vanilla/net/core/skbuff.c 2011-08-05 00:59:21.000000000 -0400
--- linux-3.0.1-tcp-nrsacks-dev/net/core/skbuff.c 2014-04-16 11:50:34.000000000 -0400
***************
*** 398,403 ****
--- 398,433 ----
  #endif
  }

+ /* UD_PEL TCP-NRSACKs:
+  * free paged data of a skb */
+ int skb_free_frags(struct sk_buff *skb)
+ {
+  int i, frags_size;
+
+  if (!skb)
+  return 0;
+
+  frags_size = 0;
+  if (skb_shinfo(skb)->nr_frags)
+  for (i = 0; i < skb_shinfo(skb)->nr_frags; i++)
+  frags_size += skb_shinfo(skb)->frags[i].size;
+
+  /* for testing */
+  //printk("%u bytes are freed\n", frags_size);
+
+  if (skb_shinfo(skb)->nr_frags)
+  for (i = 0; i < skb_shinfo(skb)->nr_frags; i++)
+  put_page(skb_shinfo(skb)->frags[i].page);
+
+  /* update corresponding skb fields */
```

```
+  skb->data_len -= frags_size;
+  skb->len -= frags_size;
+  skb->truesize -= frags_size;
+  skb_shinfo(skb)->nr_frags = 0;
+
+  return frags_size;
+ }
+
  /* Free everything but the sk_buff shell. */
  static void skb_release_all(struct sk_buff *skb)
  {
diff -rcNP linux-3.0.1-vanilla/net/ipv4/tcp_input.c linux-3.0.1-tcp-nrsacks-dev/net/ipv4/tcp_input.c
*** linux-3.0.1-vanilla/net/ipv4/tcp_input.c 2011-08-05 00:59:21.000000000 -0400
--- linux-3.0.1-tcp-nrsacks-dev/net/ipv4/tcp_input.c 2014-04-16 11:44:41.000000000 -0400
***************
*** 3235,3240 ****
--- 3235,3263 ----
   s32 ca_seq_rtt = -1;
   ktime_t last_ackt = net_invalid_timestamp();

+  /* UD_PEL TCP-NRSACKs:
+   * examine all skbs in the send buffer,
+   * free paged data of a skb if it is SACKed;
+   * skb struct and linear data part is preserved */
+
+  skb = tcp_write_queue_head(sk);
+
+  while (skb && skb != (struct sk_buff *)(&(sk)->sk_write_queue) && skb != tcp_send_head(sk)) {
+  struct sk_buff *skb_nrsacked;
+  u32 nrsacked_free_size; /* memory space freed by NR-SACKs (only paged data) */
+
+  if (TCP_SKB_CB(skb)->sacked & TCPCB_SACKED_ACKED) {
+  skb_nrsacked = skb;
+  skb = skb->next;
+  nrsacked_free_size = skb_free_frags(skb_nrsacked);
+  /* memory accounting */
+  sk->sk_wmem_queued -= nrsacked_free_size;
+  sk_mem_uncharge(sk, nrsacked_free_size);
+  }else
+  skb = skb->next;
+  }
+
+
   while ((skb = tcp_write_queue_head(sk)) && skb != tcp_send_head(sk)) {
   struct tcp_skb_cb *scb = TCP_SKB_CB(skb);
   u32 acked_pcount;
```