

# Transport Layer Reneging

Nasif Ekiz  
F5 Networks  
Seattle, Washington 98119  
n.ekiz@f5.com

Paul D. Amer  
Computer and Information Sciences Department  
University of Delaware  
Newark, Delaware 19716  
amer@cis.udel.edu

## ABSTRACT

Reneging occurs when a transport layer data receiver first selectively acks data, and later discards that data from its receiver buffer prior to delivery to the receiving application or socket buffer. Reliable transport protocols such as TCP (Transmission Control Protocol) and SCTP (Stream Control Transmission Protocol) are designed to tolerate reneging. We argue that this design should be changed because: (1) reneging is a rare event in practice, and the memory saved when reneging does occur is insignificant, and (2) by not tolerating reneging, transport protocols have the potential for improved performance as has been shown in the case of SCTP. To support our argument, we analyzed TCP traces from three different domains (Internet backbone, wireless, enterprise). We detected reneging in only 0.05% of the analyzed TCP flows. In each reneging case, the operating system was fingerprinted thus allowing the reneging behavior of Linux, FreeBSD and Windows to be more precisely characterized. The average main memory returned each time to the reneging operating system was on the order of two TCP segments. Reneging saves so little memory that it is not worth the trouble. Since reneging happens rarely and when it does happen, reneging returns insignificant memory, we recommend designing reliable transport protocols to not permit reneging.

## Categories and Subject Descriptors

C.2.5 [Local and Wide-Area Networks]: Internet – TCP; C.4 [Performance of Systems]: Measurement techniques

## General Terms

Measurement

## Keywords

OS fingerprinting, Reneging, SACK, Selective Acknowledgment, SCTP, TCP

## 1. INTRODUCTION

Transmission Control Protocol (TCP) [19] and the Stream Control Transmission Protocol (SCTP) [23] use sequence numbers and cumulative acks (ACKs) to achieve reliable data transfer. A data receiver uses sequence numbers to sort arrived

data segments. Data arriving in expected order, i.e., *ordered data*, are cumulatively ACKed to the data sender. With receipt of an ACK, the data sender assumes the data receiver accepts responsibility for delivering ACKed data to the receiving application, and the data sender deletes all ACKed data from its send buffer, potentially before that data are delivered.

The Selective Acknowledgment Option, RFC2018 [16], extends TCP's (and SCTP's) cumulative ACK mechanism by allowing a data receiver to ack arrived *out-of-order data* using selective acks (SACKs). The intent is that SACKed data need not be retransmitted during loss recovery. SACKs improve throughput when multiple losses occur within the same window [1, 4, 9].

Transport layer data reneging (simply, reneging) occurs when a data receiver first SACKs data, and later discards that data from its receiver buffer *prior* to delivery to the receiving application or socket buffer. TCP is designed to tolerate reneging. RFC2018 states: “The SACK option is advisory” and “the data receiver is permitted to later discard data which have been reported in a SACK option”. Reneging might happen, for example, when an operating system needs to recapture previously allocated memory, say to avoid deadlock, or to protect the operating system against denial-of-service attacks (DoS). As will be discussed in detail in this paper, reneging is implemented in FreeBSD, Linux, Mac OS, and Windows.

Because TCP tolerates reneging, a TCP data sender must retain copies of all transmitted data in its send buffer, even SACKed data, until they are ACKed. Then, if reneging does occur, eventually the sender will (1) timeout on the reneged data, (2) delete all SACK information, and (3) retransmit the retained copies of the reneged data. The data transfer thus remains reliable. Unfortunately, if reneging does not happen, SACKed data is wastefully stored in the send buffer until ACKed.

A similar design to tolerate reneging is adopted by SCTP. The main difference is that an SCTP data sender is designed to identify a data receiver that reneges, whereas a TCP data sender is not. When previously SACKed data are not repeatedly SACKed in the successive ack, an SCTP data sender infers reneging and marks reneged data for retransmission [23].

We argue that the transport protocol design to allow reneging should be changed because: (1) reneging is a rare event in practice, and the memory saved when reneging does occur is insignificant, and (2) by not allowing reneging, reliable transport

protocols have the potential for improved performance as has been shown in the case of SCTP [18, 26].

This paper presents a thorough investigation into renegeing to support (1). For that purpose, we herein extend an earlier model [6] to detect renegeing instances in TCP traces. Then we use the extended model to analyze over 200,000 TCP connections from three different domains to report the frequency of renegeing. For those connections that do renege, we fingerprint the OS to better understand how today's major OS deal with renegeing. The amount of potential gain (i.e., item (2)) by designing TCP to not tolerate renegeing is currently under study [2], and beyond the scope of this paper.

In Section 2, we further the motivation to detect renegeing instances and present the only past study to investigate renegeing in TCP. Then Section 3 presents our model to detect renegeing instances in TCP trace files. Section 4 presents the TCP trace analysis and results. Finally, Section 5 presents our recommendation to change the design of reliable transport protocols.

## 2. BACKGROUND AND MOTIVATION

If a transport protocol were designed not to tolerate renegeing (i.e., to be non-renegeing), a data sender would no longer need to retain copies of SACKed data in its send buffer until ACKed. In that case, the main memory allocated for the send buffer could be utilized for other data or connections.

Natarajan et al. [18] present send buffer utilization results for data transfers using non-renegeing vs. renegeing SCTP under mild (~1-2%), medium (~3-4%) and heavy (~8-9%) loss rates. For the bandwidth-delay parameters studied, the memory wasted by assuming SACKed data could be renegeed was on average ~10%, ~20% and ~30% for the given loss rates, respectively.

A non-renegeing transport protocol also can improve end-to-end application throughput. To send new data, in TCP and SCTP, a data sender is constrained by three factors: a congestion window (congestion control), an advertised receive window (flow control) and a send buffer. When the send buffer is full, no new data can be transmitted even when congestion and flow control mechanisms allow. When SACKed data are removed from the send buffer in a non-renegeing protocol, new application data can be read and potentially transmitted earlier increasing throughput.

Yilmaz et al. [26] investigate throughput improvements for non-renegeing vs. renegeing SCTP. The authors show that the throughput achieved with non-renegeing SCTP is always  $\geq$  the throughput observed with renegeing SCTP. For example, the throughput for data transfer over SCTP is improved by ~14% for a data sender with 32KB send buffer under low (~0-1%) loss rate with non-renegeing SCTP.

In summary, it has been shown if SCTP were designed to not tolerate renegeing, send buffer utilization would be always optimal, and application throughput could be improved for data transfers with constrained send buffers (send buffer < receive buffer). We

believe these SCTP results can apply to TCP as well with a modified handling of TCP's send buffer. This study is presently ongoing and not a part of this paper [2].

The key issue for this paper is – in practice, does renegeing occur or not? No one knows what percentage of connections renege. To the authors' best knowledge, the only prior study of renegeing is an MS thesis not published elsewhere [3]. The author presents a renegeing detection algorithm for a TCP data sender, and analyzes TCP traces using the detection algorithm to report frequency of renegeing. The author hypothesized that discarding the SACK scoreboard at a timeout may have a detrimental impact on a connection's ability to recover loss without unnecessary retransmissions. To decrease unnecessary retransmissions, an algorithm to detect renegeing at a TCP sender is proposed which clears the SACK scoreboard immediately upon detecting renegeing instead of waiting until a timeout. The renegeing detection algorithm compares existing SACK blocks (scoreboard) with incoming ACKs. When an ACK is advanced to the middle of a SACK block, renegeing is detected. The author indicates renegeing can be detected earlier when the TCP receiver skips previously SACKed data. In such a case, SACKs are used for renegeing detection as in our model detailed in Section 3.

Using traces, the author analyzed TCP connections with SACKs to report frequency of renegeing. Out of 1,306,646 connections analyzed, the author identified 227 connections (0.017%) as renegeed. These results suggest that renegeing is a rare event.

Our objective is to report the frequency of renegeing in today's Internet. If we observe renegeing occurs rarely or never, we will have evidence to change the basic assumptions of transport layer protocols. By designing non-renegeing transport protocols, we hypothesize that few (if any) connections will be penalized, and the large majority of non-renegeing connections will potentially benefit from better send buffer utilization and throughput.

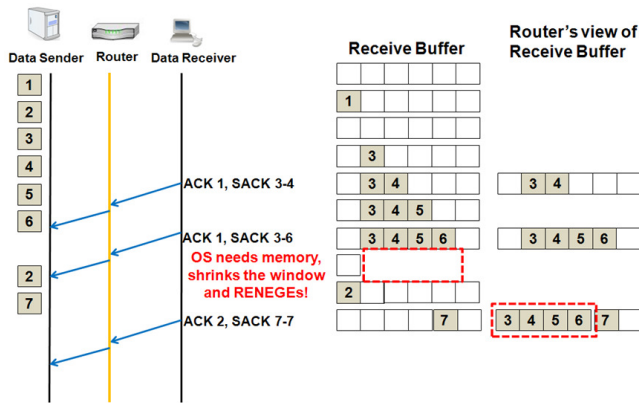
## 3. A MODEL TO DETECT RENEGING

To empirically investigate the frequency of renegeing, we present our extended model and its implementation, RenegDetectv2, to passively detect renegeing instances occurring in TCP traces.

While TCP does not support detecting renegeing at a data sender, SCTP does. In SCTP, when previously SACKed data are not repeatedly SACKed in successive acks as is specified, an SCTP data sender infers renegeing. Our initial model to detect TCP renegeing extends SCTP's renegeing detection mechanism [6].

A state of the data receiver's receive buffer is constructed at an intermediate router and updated as new acks are observed. The state consists of a cumulative ACK value (stateACK) and a list of out-of-order data blocks (stateSACK blocks) known to be in the data receiver's buffer. When an inconsistency occurs between the state of the receive buffer and a new ack, renegeing is detected. Our initial model was introduced in [6], and is now described so as to understand how and why we needed to extend it.

Figure 1 illustrates an example renegeing scenario, and how our



**Figure 1. Detecting renegeing at an intermediate router [6]**

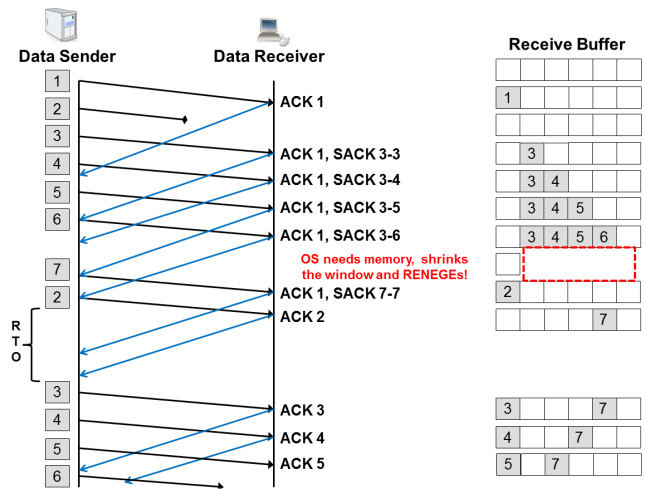
initial model located at an intermediate router detects renegeing. Three acks are monitored within a data transfer. For simplicity, data packets are not shown. Without loss of generality, the example assumes 1 byte of data is transmitted in each data packet. For each SACK X-Y, X and Y represent the left edge and right edge of the SACK, respectively.

On seeing ACK 1 SACK 3-4, our model deduces the state of receive buffer to be: ordered data 1 is delivered or deliverable to the receiving application (stateACK 1), and out-of-order data 3-4 are in the receive buffer (stateSACK 3-4). ACK 1 SACK 3-6 updates this state by adding out-of-order data 5-6 as SACKed (stateSACK 3-6). When ACK 2 SACK 7-7 is received and compared to the state of receive buffer (stateACK 1, stateSACK 3-6), an inconsistency is observed and renegeing is detected since data 3-6 are not SACKed again as they should be had renegeing not occurred.

In [6], we implemented the initial model as a tool called RenegDetect and tested it with artificial TCP flows mimicking renegeing and non-renegeing flows. RenegDetect was also verified by analyzing 100s of TCP flows from Internet traces. Initial analysis surprisingly showed that renegeing was happening frequently. On closer inspection, however, it turned out that renegeing was not happening; rather the generation of SACKs in monitored TCP implementations was incorrect according to RFC2018, wrongly giving the impression that renegeing was occurring. Some TCP implementations were generating incomplete SACKs. Sometimes SACK information that should have been sent was not. Sometimes wrong SACK information was sent. We refer to these implementations as misbehaving.

Our discovery led us to a side investigation to precisely identify five misbehaving TCP stacks. We tested 29 operating systems and found at least one misbehaving TCP stack for each of the five misbehaviors observed [7].

Discovering the TCP SACK generation misbehaviors required us to extend our model in [6] to a second version, RenegDetectv2. In addition to analyzing monitored acks, RenegDetectv2 analyzes the bidirectional flow of data, in particular, retransmissions of data, which more definitively indicate renegeing has occurred.



**Figure 2. Detecting renegeing by analyzing retransmissions**

In misbehaviors, out-of-order data are not renegeed; rather SACK information is missing or incomplete. Eventually, when the data between the ACK and the out-of-order data are received, the ACK is increased beyond the out-of-order data that seemed to have been renegeed. RenegDetectv2 concludes a misbehavior is observed (no renegeing) if no retransmissions are monitored for the out-of-order data that seemed to have been renegeed, and ACK is increased beyond the supposedly renegeed data.

With renegeing, when the data between the ACK and renegeed out-of-order data are received, the ACK would increase to the left edge of the renegeed data. Eventually, the data sender will timeout and retransmit the renegeed data. Then, the ACK would increase steadily after each retransmission. RenegDetectv2 keeps track of retransmissions for the out-of-order data that seems to have been renegeed (MISSING).

Figure 2 illustrates how RenegDetectv2 works. The example is similar to that shown in Figure 1 with the inclusion of data packets. Before packet 7 is received, the data receiver renegees and deletes out-of-order data 3-6. When packet 7 is received, ACK 1 SACK 7-7 is sent back to the data sender. When this ack is compared to the state (stateACK 1 stateSACK 3-6), an inconsistency is detected. Previously SACKed data 3-6 are not SACKed either due to renegeing or a misbehaving TCP stack. RenegDetectv2 marks data 3-6 as MISSING. The ack, ACK 2, for packet 2's fast retransmission gives the impression that renegeing happened since ACK is not increased to 7. If ACK had been increased to 7 on the receipt of packet 2, a SACK generation misbehavior (no retransmissions) would be concluded. After a retransmission timeout (RTO), the data sender retransmits packets 3-6. Since ACK increases steadily after each retransmission, a case of possible renegeing is identified.

RenegDetectv2 reports possible renegeing instances. We then analyze each possible renegeing instance by hand with Wireshark [24] to conclude if renegeing really happened. Wireshark can graph a TCP flow displaying both data and ack segments. Initially, Wireshark did not have the support to view SACK blocks. To

facilitate flow analysis, we extended Wireshark to display SACK blocks in a flow graph [25].

## 4. EMPIRICAL TRACE ANALYSIS

We now report the frequency of renegeing in TCP traces from three domains: Internet backbone (CAIDA traces), a wireless network (SIGCOMM 2008 traces), and an enterprise network (LBNL traces). In total RenegDetectv2 analyzed 202,877 TCP flows that use SACKs. In the flows, we confirmed 104 renegeing instances (~0.05%). With 95% confidence, the margin of error is 0.009% assuming that the analyzed TCP flows are independent and identically distributed (i.i.d.). That is, we estimate with 95% confidence that the true average rate of renegeing is in the interval [0.041%, 0.059%], roughly 1 flow in 2,000.

While our selection of TCP flows was random, it must be said that some characteristics suggest that the TCP flows are not i.i.d. in which case the confidence interval would be larger. For instance, on a FreeBSD host when one flow is renegeed, all other active flows are renegeed (a.k.a. global renegeing – discussed in Section 4.3). This simultaneous renegeing implies potential dependence. Similarly, TCP flows from different operating systems may not be identically distributed. A TCP flow from an OpenBSD host cannot be renegeed (thus its probability of renegeing is 0) while a FreeBSD flow can be renegeed.

For each renegeing flow, we fingerprint the operating system of the renegeing data receiver, and generalize renegeing behavior per operating system.

Trace files provided by the three domains contain thousands of TCP flows per trace. In our analysis, trace files were filtered to have a single trace file for each bidirectional TCP flow that uses SACKs. This approach served two purposes: (1) to provide renegeing traces to the research community, and (2) to be able to view a flow graph per TCP flow in Wireshark for hand analysis. Further details of processing TCP traces can be found in [8].

### 4.1 Description of Traces

The trace files from Cooperative Association for Internet Data Analysis (CAIDA) [5] are representative of wide area Internet traffic, and were collected via data collection monitors set in Equinix data centers in Chicago and San Jose, CA.

CAIDA provides 60 minute long traces for each Equinix monitor (Chicago, San Jose) per month since 2008. In our lab, we did not have enough processing capacity to filter all CAIDA traces. Instead, we processed randomly chosen 2 minute traces for each month whenever trace data was available for both directions. When we detected renegeing instances, we also processed 10 minute traces (covering the 2 minute trace) for the renegeed data receivers to analyze renegeing behavior for longer durations in more detail.

SIGCOMM traces were collected at the SIGCOMM 2008 conference, and monitored the wireless network activity during the conference [22].

Lawrence Berkeley National Laboratory (LBNL) traces characterize internal enterprise traffic recorded at a medium-sized site for five days from October, 2004 to January, 2005 [14].

## 4.2 Results

Table 1 presents the frequency of renegeing in the TCP traces for the three domains.

**Table 1. Frequency of renegeing**

Trace	Flows using SACKs	Total Renegeed Flows
CAIDA	161440	104
SIGCOMM	15683	0
LBNL	25754	0
TOTAL	202877	104

In CAIDA traces, 104 flows renegeed out of 161,440 TCP flows analyzed. These TCP traces can be downloaded [20]. In SIGCOMM and LBNL traces, no renegeing flows were detected. In [3], the author also analyzed LBNL traces and reported no instances of renegeing as we did.

We analyzed each renegeing flow in detail and categorized renegeing instances based on the OS of the data receiver. We detail renegeing instances and behavior for Linux, FreeBSD, and Windows in Sections 4.3, 4.4, and 4.5, respectively.

### 4.3 Linux Renegeing Instances

Table 2 details the TCP fingerprints (characteristics) of the five renegeing data receivers. The columns show an arbitrary host id, maximum segment size (MSS), window scale value, initial receiver window (rwnd), maximum rwnd value observed during the connection, if timestamps (TS) were used (RFC1323 [13]), and if DSACKs were used (RFC2883 [11]), respectively. We believe these data receivers were running Linux since they all exhibited the following behaviors. First, Linux implements dynamic right-sizing (DRS) where the rwnd dynamically changes based on the receiver’s estimate of the sender’s congestion window [10]. With DRS, the initial advertised rwnd of a Linux TCP is 5840 bytes and changes dynamically over the course of the connection. Second, Linux TCP supports DSACKs by default (`sysctl net.ipv4.tcp_dsack = 1`) and DSACKs were observed for all data receivers.

**Table 2. Host characteristics of Linux data receivers**

Host id	MSS (SYN)	Win Scale	Rwnd (SYN)	Rwnd (Max)	TS	DSACK
1	1460	n/a	5840	auto	no	yes
2	1460	n/a	5840	auto	no	yes
3	1460	n/a	5840	auto	no	yes
4	1460	n/a	5840	auto	no	yes
5	1460	n/a	5840	auto	no	yes

Table 3 reports the renegeing instances detected at the Linux data receivers. A total of 114 renegeing instances were observed occurring in 40 flows from five different Linux data receivers. The observation suggests that when a data receiver renegees, it tends to renege more than once within a flow, on average 2.85 times per flow.

**Table 3. Linux renegeing instances**

Host id	Reneged Flows	Reneging Instances	Avg. Reneged Bytes
1	4	9	2758
2	2	3	8273
3	28	74	1973
4	4	25	4088
5	2	3	3893
TOTAL	40	114	2715

*Definition:* We define “local renegeing” for operating systems that cause renegeing for each TCP connection independently. With local renegeing, renegeing and non-renegeing flows coexist simultaneously. We define “global renegeing” for operating systems that cause renegeing for all TCP connections simultaneously.

Linux employs local renegeing. To confirm that behavior, we analyzed renegeing times for each data receiver, and verified that renegeing instances from simultaneous flows occurred at different times. As a result, renegeing and non-renegeing connections exist in Linux simultaneously.

In [21], the authors state that renegeing in Linux is expected to happen when (a) an application is unable to read data queued up at the receive buffer, and (b) a large number of out-of-order segments are received. We confirm (a), but our analysis showed that the average amount of bytes renegeed per renegeing instance was 2715 bytes (~2 MSS PDUs.) This average is not large compared to Linux’s 87380 byte default receive buffer size (sysctl *net.ipv4.tcp\_rmem* = 4096 (min) 87380 (default) 2605056(max)). On average, only ~3% of the receive buffer was allocated to the renegeed out-of-order data. This behavior suggests that Linux renegees irrespective of out-of-order data size contrary to [21]’s claim.

#### 4.4 FreeBSD Renegeing Instances

For the two renegeing data receivers listed in Table 4, both had an initial rwnd of 65535 and used timestamps (RFC1323) by default. Table 5 lists the initial rwnd reported in SYN segments of various operating systems observed during our RFC2018 conformant SACK generation testing [7]. As the renegeing data receivers did, FreeBSD, Mac OS X and Windows 2000 all initially advertised an rwnd of 65535 bytes. The renegeing data receivers could not be running Windows 2000 because sometimes 3 or 4 SACK blocks were reported in TCP PDUs of the renegeing flows, and Windows 2000 reports at most 2 SACK blocks (Misbehavior A2) [7].

FreeBSD and Mac OS differ in the way they implement the window scale option (RFC1323). Mac OS advertises a scaled rwnd in the SYN segment. For example, if window scale option=1 for the connection, the rwnd reported in the SYN segment would be 32768 for a 65535 size rwnd. FreeBSD, on the other hand, initially advertises an rwnd of 65535 irrespective of window scale option. If the window scale option is used, say window scale=1, consecutive TCP segments would have rwnd value of 32768. During the analysis, the renegeing data receivers initially advertised an rwnd of 65535 in the SYN packet and advertised rwnds ~32K in the rest of the PDUs. Therefore, we believe these renegeing data receivers were running FreeBSD.

**Table 4. Host characteristics of FreeBSD data receivers**

Host id	MSS (SYN)	Win Scale	Rwnd (SYN)	Rwnd (Max)	TS	DSACK
1	1460	1	65535	65535	yes	no
2	1460	1	65535	65535	yes	no

**Table 5. Initial advertised rwnd of various OSes**

Operating System	Initial Advertised Rwnd (bytes)
FreeBSD 5.3-8.0	65535
Linux 2.4.18-2.6.31	5840
Mac OS X 10.6.0	65535
OpenBSD 4.2-4.7	16384
OpenSolaris 2008-2009	49640
Solaris 10	49640
Windows 2000	65535
Windows XP, Vista, 7	64240

Table 6 reports renegeing instances detected at the FreeBSD data receivers. A total of 11 renegeing instances were observed in 11 flows from two different hosts, that is, each flow renegeed exactly one time. The average bytes renegeed per renegeing instance was 3717 bytes (~2.5 MSS PDUs.) This amount of renegeed out-of-order data is insignificant (only ~5.6%) compared to FreeBSD’s 65535 byte default receive buffer size (sysctl *net.inet.tcp.recvspace*: 65536). This behavior suggests that FreeBSD renegees irrespective of out-of-order data size.

**Table 6. FreeBSD renegeing instances**

Host id	Reneged Flows	Reneging Instances	Avg. Reneged Bytes
1	1	1	4380
2	10	10	3650
TOTAL	11	11	3716

According to [12], FreeBSD employs global renegeing. To confirm this behavior, we analyzed renegeing times for the data receiver

identified with host id 2. The renegeing instances were clustered around two times: 09:19:02.0xx and 09:19:31.5yy. These clustered renegeing times confirm that FreeBSD employs global renegeing.

#### 4.5 Windows Renegeing Instances

We believe that renegeing data receivers listed in Table 7 are Windows hosts. First, all of the renegeing data receivers reported at most 2 SACK blocks, and the data receivers identified by host ids 2 and 9 reported at most 2 SACKs when it was known that at least 3 SACK blocks existed at the receiver (Misbehavior A2). Misbehavior A2 was observed only in Windows 2000, XP and Server 2003 [7]. The TCP/IP implementation for these operating systems is detailed in [15] and [17]. For the three Windows systems, the advertised rwnd is determined based on the media speed. [17] specifies that if the media speed is [1Mbps-100Mbps), rwnd is set to twelve MSS segments. If the media speed is [100Mbps-above), rwnd is set to 64KB. Only the data receivers specified with host ids 3 and 6 did not match this specification. But their maximum rwnd was set to 25\*MSS and 45\*MSS during the course of connection, respectively. Both [15] and [17] specify that Windows TCP adjusts rwnd to increments of the maximum segment size (MSS) negotiated during connection setup. This specification makes us believe those data receivers were running Windows.

Table 8 reports 75 Windows renegeing instances were observed in 53 flows from 9 different hosts, an average of 1.41 renegeing instances per renegeing flow. The average bytes renegeed per renegeing instance was 1371 bytes (~ 1 MSS PDU).

**Table 7. Host characteristics of Windows data receivers**

Host Id	MSS (SYN)	Win Scale	Rwnd (SYN)	Rwnd (Max)	TS	DSACK
1	1452	n/a	16384	17424	no	no
2	n/a	n/a	n/a	61320	no	no
3	1360	n/a	32767	34000	no	no
4	1460	n/a	65535	65535	no	no
5	1460	n/a	65535	65535	no	no
6	1452	n/a	64240	65340	no	no
7	n/a	n/a	n/a	65535	no	no
8	1460	n/a	65535	65535	no	no
9	1414	n/a	65535	65535	no	no

Since the Windows TCP/IP stack is not open-source, it is unknown if Windows employs local or global renegeing. However, the Windows renegeing instances from different flows all happened at different times suggesting that Windows employs local renegeing.

In general, only a single out-of-order segment was renegeed in the Windows renegeing instances caused by packet reordering in the network. This observation explains why the average renegeed bytes

(1371) are less than 1 MSS PDU. The consecutive out-of-order data packets received were not SACKed even though these data were known to be in the receive buffer.

**Table 8. Windows renegeing instances**

Host id	Reneged Flows	Reneging Instances	Avg. Reneged Bytes
1	1	1	98
2	1	3	2920
3	6	20	754
4	1	1	4096
5	1	1	1460
6	1	1	287
7	1	2	1965
8	1	2	3550
9	40	44	1409
TOTAL	53	75	1371

## 5. CONCLUSIONS

Trace analysis of TCP flows demonstrates that renegeing rarely occurs in practice, its frequency being in the range of one flow per 2000 (0.05%). And when renegeing does occur, memory recovered is on the order of two TCP segments (2715, 3717, and 1371 bytes for Linux, FreeBSD, and Windows, respectively), and is unlikely to help resume normal operation. Therefore, we believe new transport protocols should not permit renegeing, and any next generation version of existing reliable transport protocols (TCP, SCTP) should be designed not to permit renegeing. Additionally, results have already been shown how performance for SCTP connections can improve (and will never degrade) if renegeing is not permitted.

For example, using our trace results, let us compare TCP’s current design to tolerate renegeing with a TCP that does not support renegeing. Currently, TCP tolerates renegeing and maintains the reliable data transfer of 104 renegeing flows. If renegeing was not permitted, SACKed data would be unnecessarily stored in the TCP send buffer. The 202,773 non-renegeing flows “waste” this memory. With a revised handling of TCP’s send buffer, this memory could be used to send new data, thus better utilizing memory and potentially improving throughput [2].

For a next generation TCP, we suggest that the semantics of TCP SACKs be changed from *advisory* to *permanent* thereby prohibiting a data receiver from renegeing. In the rare event that a data receiver would need to take back memory that has been allocated to received out-of-order data, we propose that the data receiver must RESET the transport connection. With this change, 104 renegeing flows would be terminated (i.e., RESET). In these cases, termination is not a penalty. If the renegeing was due to memory stress, given that little memory is recovered by renegeing, RESETing the existing connections would help the stressed host better since more memory is reclaimed back by terminating

connections compared to renegeing. If the renegeing was due to a DoS attack, then RESEtting the connection is again a better response than renegeing.

On the other hand, 202,773 non-renegeing flows benefit with better send buffer utilization, and could potentially achieve increased throughput. Note that increased TCP throughput is only possible for data transfers with constrained send buffers (assuming asymmetric buffer sizes (send buffer < receive buffer)), and needs modifications in TCP's send buffer management [2]. Increasing the data sender's send buffer size to the data receiver's receive buffer size makes throughput no longer limited by the send buffer size. However, with this change, a host can support fewer TCP connections. For future study, we propose to investigate the tradeoff of a host supporting fewer connections as a result of increased send buffer size vs. supporting more connections by not allowing renegeing.

We believe current protocols (TCP, SCTP) need not tolerate renegeing, but recognize that the hurdle of deploying any change to these existing protocols can only be justified by a significant performance gain. Our current efforts [2] are to quantify those gains.

The primary contribution of this work is empirical evidence arguing that *future transport protocols should be not designed to permit renegeing*. There is simply no gain by renegeing, and with small send buffers, there can be a performance penalty. Simply RESEtting connections that might need to be reneged is a better solution. If a next generation TCP, SCTP is proposed, removing the tolerance of renegeing should be one of several incorporated improvements.

## 6. ACKNOWLEDGMENTS

The authors thank A. Rahman, F. Yang, and J. Leighton for their valuable discussions and comments. The authors also thank the anonymous reviewers for their valuable improvements.

## 7. REFERENCES

- [1] Allman, M., Hayes, C., Kruse H., and Ostermann, S. TCP Performance over Satellite Links. In *Proc 5th Int'l Conf on Telecomm. Systems*. (Mar. 1997)
- [2] Amer, P., and Ekiz, N. Improving TCP Throughput by Not Tolerating Reneging (in progress)
- [3] Blanton, J. T. A Study of TCP SACK State Lifetime Validity. MS Thesis. (Nov. 2008). Ohio Univ.
- [4] Bruyeron, R., Hemon, B., and Zhang, L. Experimentations with TCP Selective Ack. *SIGCOMM Comput. Commun. Rev.* 28, 2 (Apr. 1998), 54-77. DOI=10.1145/279345.279350
- [5] CAIDA Internet Data – Passive Data Sources. [www.caida.org/data/passive](http://www.caida.org/data/passive)
- [6] Ekiz, N. and Amer, P. A Model for Detecting Transport Layer Data Reneging. In *8th Int'l Workshop on Protocols for Future, Large-Scale & Diverse Network Transports* (Nov. 2010). PFLDNeT '10. [www.cis.udel.edu/~amer/PEL/poc/pdf/PFL.pdf](http://www.cis.udel.edu/~amer/PEL/poc/pdf/PFL.pdf)
- [7] Ekiz, N., Rahman, A., and Amer, P. Misbehaviors in TCP SACK Generation. *SIGCOMM Comput. Commun. Rev.* 41, 2 (Apr. 2011), 16-23. DOI=10.1145/1971162.1971165
- [8] Ekiz, N. 2012. *Transport Layer Reneging*. PhD Dissertation. University of Delaware.
- [9] Fall, K. and Floyd, S. Simulation-based Comparisons of Tahoe, Reno and SACK TCP. *SIGCOMM Comput. Commun. Rev.* 26, 3 (Jul. 1996), 5-21. DOI=10.1145/235160.235162
- [10] Fisk, M. and Feng, W. Dynamic Right-Sizing: TCP Flow-Control Adaptation. In *Proc 14<sup>th</sup> ACM/IEEE Supercomputing Conf.* (Nov. 2001)
- [11] Floyd, S., Mahdavi, J., Mathis, M., and Podolsky, M. An Extension to the Selective Acknowledgement (SACK) Option for TCP. RFC2883. (Jul. 2000)
- [12] FreeBSD TCP Implementation. [www.freebsd.org/cgi/cvsweb.cgi/src/sys/netinet](http://www.freebsd.org/cgi/cvsweb.cgi/src/sys/netinet).
- [13] Jacobson, V., Braden, R., and Borman, D. TCP Extensions for High Performance. RFC1323. (May 1992)
- [14] LBNL/ISCI Enterprise Tracing Project. [www.icir.org/enterprise-tracing](http://www.icir.org/enterprise-tracing)
- [15] MacDonald, D. and Barkley, W. *Microsoft Windows 2000 TCP/IP Implementation Details*. Microsoft. [technet.microsoft.com/en-us/library/bb726981.aspx](http://technet.microsoft.com/en-us/library/bb726981.aspx)
- [16] Mathis, M., Mahdavi, J., Floyd, S., and Romanow, A. TCP Selective Ack Options. RFC2018. (Oct. 1996)
- [17] Microsoft. 2003. *Microsoft Windows Server 2003 TCP/IP Implementation Details*. (Jun. 2003)
- [18] Natarajan, P., Ekiz, N., Yilmaz, E., Amer, P., Iyengar, J., Stewart, R. Non-Renegable SACKs for SCTP. In *IEEE Int'l Conf on Network Protocols*. (Oct. 2008), 187-196. DOI=dx.doi.org/10.1109/ICNP.2008.4697037
- [19] Postel, J. 1981. Transmission Control Protocol. RFC793 (Sep. 1981).
- [20] Reneged TCP flow traces. [www.cis.udel.edu/~amer/PEL/reneging\\_traces.tar](http://www.cis.udel.edu/~amer/PEL/reneging_traces.tar)
- [21] Seth, S. and Ajaykumar, V. *TCP/IP Architecture, Design and Implementation in Linux*, John Wiley & Sons, 2008
- [22] SIGCOMM 2008 Traces. [www.cs.umd.edu/projects/wifidelity/sigcomm08\\_traces](http://www.cs.umd.edu/projects/wifidelity/sigcomm08_traces)
- [23] Stewart, R. Stream Control Transmission Protocol. RFC4960. (Sep. 2007)
- [24] Wireshark. [www.wireshark.org](http://www.wireshark.org)
- [25] Wireshark patch to view SACKs: [www.cis.udel.edu/~amer/PEL/Wire\\_TCP\\_patch.tar](http://www.cis.udel.edu/~amer/PEL/Wire_TCP_patch.tar)
- [26] Yilmaz, E., Ekiz, N., Natarajan, P., Amer, P., Leighton, J., Baker, F., and Stewart, R. Throughput Analysis of Non-Renegable SACKs for SCTP. *Comput. Commun.* 33, 16 (Oct. 2010), 1982-1991. DOI=dx.doi.org/10.1016/j.comcom.2010.06.028.