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 and SCTP 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 tolerate 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 reliable transport throughput when multiple losses occur within the same window [1, 4, 9].

Transport layer data reneging (or 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). Reneging is possible in FreeBSD, Linux, Mac OS, Solaris and Windows.

Because TCP is designed to tolerate 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 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, reliable transport protocols have the potential for improved performance as has been shown in the case of SCTP [18, 26].

This paper's goal is to present a thorough investigation into reneging to support (1). For that purpose, we develop a model to detect reneging instances in TCP traces and analyze traces from three different domains using our model to report the frequency of reneging. The amount of potential gain by designing TCP to not tolerate reneging is currently under study [2], and beyond the scope of this paper.

In Section 2, we further the motivation to detect reneging instances. Then Section 3 presents our model to detect reneging instances in TCP trace files. Section 4 presents the TCP trace analysis and results. Section 5 identifies the only past study to investigate reneging in TCP. Finally, Section 6 presents our recommendation to change the design of reliable transport protocols.

2. MOTIVATION TO NOT TOLERATE RENEGING

If a transport protocol were designed not to tolerate reneging (i.e., to be non-reneging), a data sender would no longer need to retain copies of SACKed data in its send buffer until ACKed. Just as with ACKed data on the receipt of an ACK, if reneging was not allowed, SACKed data could be removed from the send buffer immediately on the receipt of a SACK. In that case, the main memory allocated for the send buffer could be utilized for other data.

Natarajan et al. [18] present send buffer utilization results for data transfers using non-reneging vs. reneging 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 reneged was on average ~10%, ~20% and ~30% for the given loss rates, respectively.

A non-reneging 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-reneging protocol, new application data can be read and potentially transmitted.

Yilmaz et al. [26] investigate throughput improvements for non-reneging vs. reneging SCTP. The authors show that the throughput achieved with non-reneging SCTP is always $\geq$ the throughput observed with reneging 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-reneging SCTP.

In summary, it has been shown if SCTP were designed to not tolerate reneging, 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 reneging occur or not? No one knows what percentage of connections renege. Our objective is to report the frequency of reneging in today's Internet. If we observe reneging occurs rarely or never, we will have evidence to change the basic assumptions of transport layer protocols. By designing non-reneging transport protocols, we hypothesize that few (if any) connections will be penalized, and the large majority of non-reneging connections will potentially benefit from better send buffer utilization and increased throughput.

3. A MODEL TO DETECT RENEGING

To empirically investigate the frequency of reneging, we present a model and its implementation, RenegDetect, to passively detect reneging instances occurring in TCP traces.

While TCP does not support detecting reneging 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 reneging. Our model to detect TCP reneging is based on SCTP's reneging detection mechanism.

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 receive buffer. When an inconsistency occurs between the state of the receive buffer and a new ack, reneging is detected. The model is fully detailed in [6].

Figure 1 illustrates an example reneging scenario, and how our model located at an intermediate router detects reneging. Figure 1 shows a data transfer where three acks are monitored. 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.

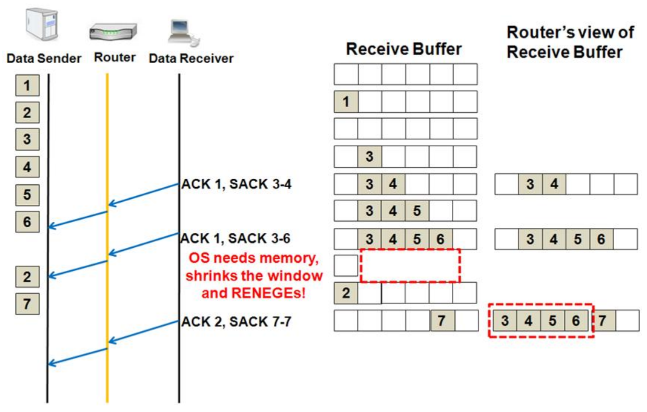

Figure 1. Detecting reneging at an intermediate router

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 reneging is detected since data 3-6 are not SACKed again as they should be had reneging not occurred.

We implemented the model as a tool called RenegDetect and tested RenegDetect with artificial TCP flows mimicking reneging and non-reneging flows. RenegDetect was also verified by analyzing 100s of TCP flows from Internet traces. Initial analysis surprisingly showed that reneging was happening frequently. On closer inspection, however, it turned out that reneging was not happening; rather the generation of SACKs in monitored TCP implementations was incorrect according to RFC2018, wrongly giving the impression that reneging 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 led us to extend RenegDetect. In addition to analyzing monitored acks, RenegDetect was extended to analyze the bidirectional flow of data, in particular, retransmissions of data, which more definitively indicate reneging has occurred.

In misbehaviors, out-of-order data are not reneged; 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 reneged. We conclude a misbehavior is observed (no reneging) if no retransmissions are monitored for the out-of-order data that seemed to have been reneged, and ACK is increased beyond the supposedly reneged data.

On the other hand, with reneging, when the data between the ACK and reneged out-of-order data are received, the ACK would increase to the left edge of the reneged data. Eventually, the data sender will timeout and retransmit the reneged data. Then, the ACK would increase steadily after each retransmission. The updated RenegDetect v2, keeps track of retransmissions for the out-of-order data that seems to have been reneged (MISSING).

Figure 2 illustrates how to detect reneging by analyzing retransmissions. The example is similar to that shown in Figure 1 with the inclusion of data packets. Before packet 7 is received, the data receiver reneges 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 again due to possible reneging or a misbehaving TCP stack. RenegDetect v2 marks data 3-6 as MISSING. The ack, ACK 2, for packet 2's fast retransmission gives the impression that reneging 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 reneging is identified

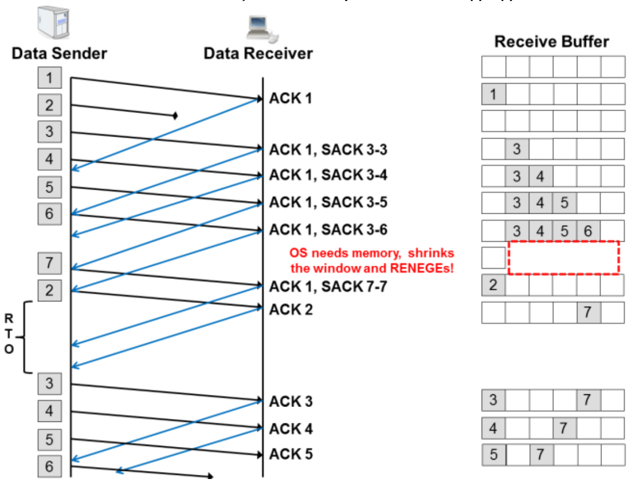

Figure 2. Detecting reneging by analyzing retransmissions

RenegDetect v2 reports possible reneging instances. We then analyze each possible reneging instance by hand with Wireshark [24] to conclude if reneging 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 reneging in TCP traces from three domains: Internet backbone (CAIDA traces), a wireless network (SIGCOMM 2008 traces), and an enterprise network (LBNL traces). In total we analyzed 202,877 TCP flows that use SACKs. In the flows, we confirmed 104 reneging 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 reneging 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 reneged, all other active flows are reneged (a.k.a. global reneging – discussed in Section 4.3). This simultaneous reneging implies potential dependence. Similarly, TCP flows from different operating systems may not be identically distributed. A TCP flow from an OpenBSD host cannot be reneged (thus its probability of reneging is 0) while a FreeBSD flow can be reneged.

For each reneging flow, we fingerprint the operating system of the reneging data receiver, and generalize reneging 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 reneging 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].

RenegDetect v2 accepts a TCP trace file as an input and analyzes a TCP flow using our model detailed in Section 3. RenegDetect v2 logs possible reneging flows (and each individual instance per flow) during the trace analysis. Possible reneging instances are inspected by hand using Wireshark to conclude reneging or not.

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 reneging instances, we also processed 10 minute traces (covering the 2 minute trace) for the reneged data receivers to analyze reneging 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 reneging in the TCP traces for the three domains.

Table 1. Frequency of reneging

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

In CAIDA traces, 104 flows reneged out of 161,440 TCP flows analyzed. These TCP traces can be downloaded [20]. In SIGCOMM and LBNL traces, no reneging flows were detected.

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

4.3 Linux Reneging Instances

Table 2 details the TCP fingerprints (characteristics) of the five reneging 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. Linux reneging 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

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

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

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

In [21], the authors state that reneging 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 reneged per reneging 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 reneged out-of-order data. This behavior suggests that Linux reneges irrespective of out-of-order data size contrary to [21]'s claim.

4.4 FreeBSD Reneging Instances

For the two reneging 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 reneging data receivers did, FreeBSD, Mac OS X and Windows 2000 all initially advertised an rwnd of 65535 bytes. The reneging data receivers could not be running Windows 2000 because sometimes 3 or 4 SACK blocks were reported in TCP PDUs of the reneging 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 reneging 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 reneging data receivers were running FreeBSD.

Table 6 reports reneging instances detected at the FreeBSD data receivers. A total of 11 reneging instances were observed in 11 flows from two different hosts, that is, each flow reneged exactly one time. The average bytes reneged per reneging instance was 3717 bytes (~2.5 MSS PDUs.) This amount of reneged 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 reneges irrespective of out-of-order data size.

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. FreeBSD reneging 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 reneging. To confirm this behavior, we analyzed reneging times for the data receiver identified with host id 2. The reneging instances were clustered around two times: 09:19:02.0xx and 09:19:31.5yy. These clustered reneging times confirm that FreeBSD employs global reneging.

4.5 Windows Reneging Instances

We believe that reneging data receivers listed in Table 7 are Windows hosts. First, all of the reneging 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 reneging instances were observed in 53 flows from 9 different hosts, an average of 1.41 reneging instances per reneging flow. The average bytes reneged per reneging 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

Table 8. Windows reneging 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

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

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

5. RELATED RESEARCH

To the authors' best knowledge, the only prior study of reneging is an MS thesis not published elsewhere [3]. The author presents a reneging detection algorithm for a TCP data sender, and analyzes TCP traces using the detection algorithm to report frequency of reneging. 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 reneging at a TCP sender is proposed which clears the SACK scoreboard immediately upon detecting reneging instead of waiting until a timeout. The reneging detection algorithm compares existing SACK blocks (scoreboard) with incoming ACKs. When an ACK is advanced to the middle of a SACK block, reneging is detected. The author indicates reneging can be detected earlier when the TCP receiver skips previously SACKed data. In such a case, SACKs are used for reneging detection as in our model.

Using real traces, the author analyzed TCP connections with SACKs to report frequency of reneging. Out of 1,306,646 connections analyzed, the author identified 227 connections (0.017%) as reneged. These results support the conclusion that reneging is a rare event. The author also analyzed LBNL traces and reported no instances of reneging as we did.

6. CONCLUSIONS

Trace analysis of TCP flows demonstrates that reneging rarely occurs in practice, its frequency being in the range of one flow per 2000 (0.05%). And when reneging does occur, relatively little memory is recovered.

Since reneging is rare and little memory is regained when reneging does occur, we believe reliable transport protocols (e.g., TCP, SCTP) should be designed not to tolerate reneging. Results have already been shown how performance for SCTP connections can improve (and will never degrade) if reneging is not tolerated.

Using our trace results, let us compare TCP's current design to tolerate reneging with a TCP that does not support reneging. Currently, TCP tolerates reneging and maintains the reliable data transfer of 104 reneging flows. If reneging could not happen, SACKed data are unnecessarily stored in the TCP send buffer. The 202,773 non-reneging 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 the connection's throughput [2].

We suggest that the current semantics of TCP SACKs be changed from *advisory* to *permanent* thereby prohibiting a data receiver from reneging. 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 reneging flows would be penalized by termination. On the other hand, 202,773 non-reneging flows could potentially benefit better send buffer utilization and 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].

Initially, reneging was thought as a utility mechanism to help an operating system reclaim main memory under dangerous low-memory situations. In trace analysis, we found that the average main memory returned to a reneging operating system per reneging instance was on the order of two TCP segments (2715, 3717, and 1371 bytes for Linux, FreeBSD, and Windows, respectively). Reclaiming such a small amount of memory does not seem worth the trouble, i.e., it is unlikely to help resume normal operation.

7. ACKNOWLEDGMENTS

The authors thank Abuthahir Habeeb Rahman, Fang Yang, and Jonathan Leighton for their valuable discussions and comments.

8. REFERENCES

[1] Allman, M., Hayes, C., Kruse H., and Ostermann, S. 1997. TCP Performance over Satellite Links. In *Proceedings of the 5th International Conference on Telecommunication Systems*. (Mar. 1997).

[2] Amer, P. D, and Ekiz, N., Improving TCP Throughput by Not Tolerating Reneging (in progress)

[3] Blanton, J. T. 2008. A Study of Transmission Control Protocol Selective Acknowledgement State Lifetime Validity. Master of Science Thesis. (Nov. 2008). Ohio University.

[4] Bruyeron, R., Hemon, B., and Zhang, L. 1998. Experimentations with TCP Selective Acknowledgment. *SIGCOMM Comput. Commun. Rev*. 28, 2 (Apr. 1998), 54-77. DOI=10.1145/279345.279350 http://doi.acm.org/10.1145/279345.279350.

[5] CAIDA Internet Data – Passive Data Sources. www.caida.org/data/passive.

[6] Ekiz, N. and Amer, P. D. 2010. A Model for Detecting Transport Layer Data Reneging. In *the 8th International Workshop on Protocols for Future, Large-Scale & Diverse Network Transports* (Lancaster, PA, November 28-29 2010). PFLDNeT '10.

[7] Ekiz, N., Rahman, A. H., and Amer, P. D. 2011. Misbehaviors in TCP SACK Generation. *SIGCOMM Comput. Commun. Rev.* 41, 2 (April 2011), 16-23. DOI=10.1145/1971162.1971165 http://doi.acm.org/10.1145/1971162.1971165.

[8] Ekiz, N. 2012. *Transport Layer Reneging*. Doctoral Dissertation. University of Delaware.

[9] Fall, K. and Floyd, S. 1996. Simulation-based Comparisons of Tahoe, Reno and SACK TCP. *SIGCOMM Comput. Commun. Rev.* 26, 3 (July 1996), 5-21. DOI=10.1145/235160.235162 http://doi.acm.org/10.1145/235160.235162.

[10] Fisk, M. and Feng, W. 2001. Dynamic Right-Sizing: TCP Flow-Control Adaptation. In *Proceedings of the 14th ACM/IEEE Supercomputing Conference*. (Nov. 2001).

[11] Floyd, S., Mahdavi, J., Mathis, M., and Podolsky, M. 2000. 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.

[13] Jacobson, V., Braden, R., and Borman, D. 1992. TCP Extensions for High Performance. RFC1323. (May 1992).

[14] LBNL/ISCI Enterprise Tracing Project. 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.

[16] Mathis, M., Mahdavi, J., Floyd, S., and Romanow, A. 1996. TCP Selective Acknowledgment 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. D., Iyengar, J., and Stewart, R. 2008. Non-Renegable Selective Acknowledgments (NR-SACKs) for SCTP. In *IEEE International Conference on Network Protocols*. (October 2008), 187-196. DOI= http://dx.doi.org/10.1109/ICNP.2008.4697037.

[19] Postel, J. 1981. Transmission Control Protocol. RFC793. (Sep. 1981).

[20] Reneged TCP flows traces. www.cis.udel.edu/~amer/PEL/reneging_traces.tar.

[21] Seth, S. and Ajaykumar, V. M. 2008. *TCP/IP Architecture, Design and Implementation in Linux*, John Wiley & Sons.

[22] SIGCOMM 2008 Traces, www.cs.umd.edu/projects/wifidelity/sigcomm08_traces

[23] Stewart, R. 2007. Stream Control Transmission Protocol. RFC4960. (Sep. 2007).

[24] Wireshark. www.wireshark.org.

[25] Wireshark patch to view SACKs: www.cis.udel.edu/~amer/PEL/Wireshark_TCP_flowgraph_patch.tar.

[26] Yilmaz, E., Ekiz, N., Natarajan, P., Amer, P. D., Leighton, J. T., Baker, F., and Stewart, R. R. 2010. Throughput Analysis of Non-Renegable Selective Acknowledgments (NR-SACKs) for SCTP. *Comput. Commun*. 33, 16 (Oct. 2010), 1982-1991. DOI= http://dx.doi.org/10.1016/j.comcom.2010.06.028.