TRANSPORT LAYER RENEGING

by

Nasif Ekiz

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

Fall 2012

TRANSPORT LAYER RENEGING

by

Nasif Ekiz

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

Approved: ______________________________
Babatunde A. Ogunnaike, Ph.D.
Interim Dean of the College of Engineering

Approved: ______________________________
Charles G. Riordan, Ph.D.
Vice Provost for Graduate and Professional Education

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

Signed: ______________________________

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

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

Signed: ______________________________

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

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

Signed: ______________________________

Martin Swany, Ph.D.
Member of dissertation committee

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

Signed: ______________________________

Janardhan R. Iyengar, Ph.D.
Member of dissertation committee

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

Signed: ____

Frederick J. Baker, Cisco Fellow
Member of dissertation committee

ACKNOWLEDGMENTS

I consider myself very lucky for having Paul D. Amer as my advisor. He has always been supportive when I have had hard-times in life and research, especially in my first years. With his help and patience, I was able to overcome all the difficulties I encountered. As an advisor, he taught me to present complex ideas in a simple and clear way. His enthusiasm for teaching and research gave me the strength to finish this dissertation. I believe I could not ask more from an advisor. Thanks a lot Prof. Amer.

I would like to thank my dissertation committee: Prof. Adarshpal Sethi, Prof. Martin Swany, Prof. Janardhan Iyengar, and Fred Baker for their reviews, constructive suggestions and criticisms that helped to improve this dissertation.

During my years at the University of Delaware, I worked with a lot of great people at the Protocol Engineering Lab (PEL). Preethi Natarajan was my mentor. She helped me and improved my ability to conduct research in many ways. Thanks a lot, Preethi. Jon Leighton and Joe Szymanski always provided invaluable comments and helpful discussions when I was stuck on a problem. Abuthahir Rahman Habeeb, Fan Yang, Aasheesh Kolli, Ersin Ozkan, and Varun Notibala contributed to this dissertation by running experiments and extending network monitoring tools. Thank you all. In addition, I am very grateful to Armando Caro, Janardhan Iyengar and Randall Stewart for their guidance whenever I faced problems with ns-2 and FreeBSD stack.

This dissertation is dedicated to my mother, Meryem Ekiz, my father, Yusuf Ekiz, and my brother, Kemal Ekiz. Without their endless support, this dream would not have come true.

A very special thanks goes to Kristina Curtis, not only for correcting my grammatical errors but also for being very supportive and patient with me during my stressful times while writing this dissertation. You are the best, canim!

Newark is a very special city for me. What made Newark really special is the close friends I met here. Bahadir Bastas, Yiannis Bourmpakis, Levent Colak, Gokhan and Juliette Tolay, Mehmet and Ozge Uygur, and Hasan Yonten made Newark more fun and enjoyable. Thank you all.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Reneging occurs when a data receiver SACKs data, and later discards that data from its receiver buffer prior to delivering it to the receiving application. Today's reliable transport protocols (TCP, SCTP) are designed to tolerate data reneging. I argue that this design assumption is wrong based on our hypothesis that reneging rarely if ever occurs in practice. To support this argument, this dissertation provides the literature's first comprehensive analysis of reneging. We investigate (1) the instances, (2) causes and (3) effects of reneging in today's Internet.

For (1), this dissertation proposes a model to detect reneging instances. The model builds upon the way an SCTP data sender detects reneging. A state of the data receiver's receive buffer is constructed at an intermediate router and updated as new acks are observed. When an inconsistency occurs between the state of the receive buffer and a new ack, reneging is detected. We implemented the proposed model as a tool called RenegDetect v1. While verifying RenegDetect v1 with real TCP flows, we discovered that some TCP implementations were generating SACKs incompletely under some circumstances giving a false impression that reneging was happening. Our discovery led us to a side investigation to precisely identify five misbehaving TCP stacks observed in the Internet (CAIDA) traces. For that, we designed a methodology and verified RFC2018-conformant SACK generation on 29 TCP stacks for a wide range of OSes. We found at least one misbehaving TCP stack for the five misbehaviors observed during the verification of RenegDetect v1 and concluded that while simple in concept, SACK handling is complex to implement.

To identify reneging instances more accurately and distinguish them from SACK generation misbehaviors, we updated RenegDetect v1 to v2 to better analyze the flow of data, in particular, to analyze data retransmissions which are a more definitive indication that reneging happened. To report the frequency of reneging in trace data, traces from three domains were analyzed: Internet backbone, a wireless network, and an enterprise network. Contrary to our initial expectation that reneging is an extremely rare event, trace analysis demonstrated that reneging does happen. We analyzed 202,877 TCP flows using SACKs from the three domains. In the flows, we confirmed 104 reneging instances (0.05%). With 95% statistical confidence, we report that the true average rate of reneging is in the interval [0.041%, 0.059%], roughly 1 flow in 2000.

For the reneging instances that were found, the operating system of the data receiver was identified thus allowing the reneging behavior of Linux, FreeBSD and Windows hosts to be more precisely characterized.

Since TCP is designed to tolerate reneging, SACKed data are unnecessarily stored in the send buffer wasting operating system resources when reneging does not happen. Since reneging does happen rarely (less than 1 flow per 1000), we recommend that TCP should be changed to not tolerate reneging by (a) changing the semantics of SACKs from being *advisory* to *permanent* and (b) RESETing (terminating) a connection if a data receiver does have to take back memory that has been allocated to received out-of-order data. With this recommended change, the send buffer is better utilized for the large majority of flows that do not renege.

In trace analysis, we also found that the average main memory returned to a reneging operating system per reneging instance was on the order of 2 TCP segments

(2715, 3717, and 1371 bytes for Linux, FreeBSD, and Windows operating systems, respectively.) This average amount of main memory reclaimed back to the operating system seems relatively insignificant. I argue that reneging to save so little memory is not worth the trouble. Reclaiming such an amount of memory to an operating system is unlikely to help resume normal operation.

The causes of reneging (2) were identified by analyzing TCP stacks of popular operating systems with reneging support. Our investigation revealed that five popular operating systems (FreeBSD, Linux (Android), Apple's Mac OS X, Oracle's Solaris and Microsoft's Windows Vista+) can renege. Initially, reneging was expected to happen on operating systems that go low on main memory to help the operating system to resume normal operation. Surprisingly, we discovered that reneging also is used as a protection mechanism against Denial of Service (DoS) attacks (Solaris and Windows Vista/7). We concluded that reneging is a common mechanism implemented in many of today's popular operating systems.

To investigate the consequences of reneging (3), a tool, CauseReneg, to cause a remote host to renege was designed, and used to force FreeBSD, Solaris, and Windows Vista victims to renege. CauseReneg achieves its goal by exhausting a victim's resources by sending out-of-order data using multiple TCP connections. For an operating system (e.g., FreeBSD) starving for memory, we demonstrated that reneging alone cannot help the system to resume normal operation. Therefore, we recommend that reneging support should be turned off for systems using reneging as a mechanism to reclaim memory to resume normal operation. For operating systems using reneging to protect against DoS attacks, reneging appears to be a useful mechanism. We argue that a better approach would be to RESET a connection under attack instead of reneging since terminating the connection would release all of the resources held. For example, in FreeBSD, reneging would reclaim at most 64 Kbytes (the default receive buffer size) per connection while terminating a connection would release ~3MB of memory. By RESETing a connection, the victim's system resources are better utilized.

Chapter 1

INTRODUCTION

1.1 Proposed Research

This dissertation investigates data reneging within the transport layer. Data reneging occurs when a data receiver buffers and selectively acknowledges out-of-order received data, and then purges that data from its receive buffer without delivering the data to the receiving application. Today's reliable transport protocols Transmission Control Protocol (TCP) [RFC793] and Stream Control Transmission Protocol (SCTP) [RFC4960] are designed to tolerate reneging. This dissertation argues this design assumption is wrong. To develop and support this argument, this dissertation investigates the instances, causes and effects of data reneging in today's Internet.

1.2 Definitions and Problem Statement

1.2.1 Transport Layer "Shrinking the Window" and "Reneging"

Data reneging is a transport layer behavior of which little is known: its frequency of occurrence, its causes, its effects, etc. This section discusses data reneging in more detail and motivates the study of data reneging in transport protocols such as TCP and SCTP.

TCP specifies sequence numbers and cumulative acknowledgments (ACKs) to help achieve reliable data transfer. A TCP data receiver uses sequence numbers to sort arrived data segments. Data arriving in expected order, i.e., *ordered data*, are acknowledged to the data sender via cumulative ACKs. The data receiver accepts responsibility of delivering ACKed data to the receiving application. Thus the data sender can safely delete all cumulatively ACKed data from its send buffer, even before these data are delivered to the receiving application.

The data receiver stores incoming data segments in a receive buffer. The receive buffer consists of two types of data: ordered data which have been cumulatively ACKed but not yet delivered to the application, and out-of-order data caused by loss or reordering in the network. A correct TCP data receiver implementation is not allowed to delete cumulatively ACKed data without first delivering these data to the receiving application since the data sender removes ACKed data from its send buffer.

Related to reneging is a behavior known as *shrinking the window*. For purposes of flow control, a data receiver advertises a receive window (rwnd) which specifies the amount of available buffer space at the data receiver. As a means of *flow control*, a data receiver constrains a data sender to have at most an rwnd of data outstanding. A TCP data receiver is technically allowed to advertise a window, and later advertise a smaller window. This shrinking the window behavior while permitted by the TCP specification is strongly discouraged [RFC793]. When a data receiver shrinks its window, no ACKed data are actually deleted from the receive buffer, only advertised empty buffer space is retracted.

TCP's Selective Acknowledgment Option (SACK) [RFC2018] extends TCP's cumulative ACK mechanism by introducing SACKs. SACKs allow a data receiver to acknowledge arrived out-of-order data to the data sender. The intention is that

SACKed data do not need to be retransmitted during loss recovery. The data receiver informs the data sender of out-of-order data by including SACK(s) in the TCP segment's options field.

Data receiver reneging (or simply, *reneging*) occurs when a data receiver SACKs data, and later discards these data from its receive buffer prior to delivering these data to the receiving application (or socket buffer). TCP is designed to tolerate reneging. Specifically [RFC2018] states that: "*The SACK option is advisory, in that, while it notifies the data sender that the data receiver has received the indicated segments, the data receiver is permitted to later discard data which have been reported in a SACK option*". As is shrinking the window, reneging also is strongly discouraged but permitted when, for example, an operating system needs to recapture previously allocated receive buffer memory for another process, say to avoid deadlock.

Because TCP is designed to tolerate possible reneging by a data receiver, a TCP data sender must keep copies of all transmitted data segments in its send buffer, even SACKed data, until cumulatively ACKed. If reneging does happen, a copy of the reneged data exists and can be retransmitted to complete the reliable data transfer. Inversely if reneging does not happen, SACKed data are unnecessarily stored in the send buffer until cumulatively ACKed.

Reneging is more serious than shrinking the window. Note that while out-of-order data are deleted from the receive buffer when reneging occurs, no removal of out-of-order data occurs with shrinking the window. A TCP data sender needs a mechanism (and its associated overhead) to deal with reneging while no extra mechanism is needed to handle shrinking the window.

This dissertation investigates if reneging actually occurs in the current Internet. If reneging never occurs, transport protocols have no need to manage the event and current TCP and SCTP implementations can be improved. Further, if reneging occurs rarely, we believe the current handling of reneging in transport protocols can be improved.

To further motivate the study of reneging, we need to understand the potential gains for a transport protocol that does not tolerate reneging. For that, we first explain Non-Renegable Selective Acknowledgments (NR-SACKs).

1.2.2 Non-Renegable Selective Acknowledgments (NR-SACKs)

The Non-Renegable Selective Acknowledgment (NR-SACKs) is a new acknowledgment mechanism for SCTP [Ekiz 2011a]. With an NR-SACK extension, an SCTP data receiver takes responsibility for non-renegable data (NR-SACKed), and, an SCTP data sender needs not to retain copies of NR-SACKed data in its send buffer until cumulatively ACKed. NR-SACKed data can be removed from the send buffer immediately on the receipt of the NR-SACK.

In SCTP, non-renegable data are possible in three ways.

(1) SCTP offers an unordered delivery service in which data marked UNORDERED can be delivered to the receiving application immediately even if the data are out-of-order according to the transport sequence number (TSN). After UNORDERED data are delivered to the application, they are by definition non-renegable.

(2) SCTP provides a multistream delivery service in which each stream is logically independent, and data received in-order within a stream can be delivered to the application. In multistream applications, data delivered to the application are non-

renegable, even though these data are out-of-order within the SCTP association (SACKed). Note that out-of-order data which are also out-of-order within a stream are renegable.

(3) It is possible to make all out-of-order data non-renegable in both TCP and SCTP with operating system support. Some operating systems allow turning reneging on and off. When reneging is off, the operating system *guarantees* not to renege on out-of-order data. In FreeBSD [Freebsd], for example, the sysctl variable *net.inet.tcp.do_tcpdrain* (a mechanism to get/set kernel state) can be used to turn reneging off for TCP. This variable is on by default. Analogously, the sysctl variable *net.inet.sctp.do_sctp_drain* is provided for SCTP. When reneging is turned off, all out-of-ordered data become non-renegable.

NR-SACKed data are released from a data sender's send buffer immediately. With NR-SACKs, only renegable (necessary) data reside in the send buffer, while with SACKs both renegable and non-renegable (unnecessary) data are kept. As a result, memory allocated for the send buffer is better utilized with NR-SACKs. [Natarajan 2008b] presents send buffer utilization results for unordered data transfers over SCTP under mild (~1-2%), medium (~3-4%) and heavy (~8-9%) loss rates using NR-SACKs vs. SACKs. For the bandwidth-delay parameters studies with SACKs, the memory wasted by keeping copies of non-renegable data is on average ~10%, ~20% and ~30% for the given loss rates, respectively.

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

[Yilmaz 2010] shows that the throughput achieved with NR-SACKs is always $\geq$ throughput observed with SACKs. For example, using NR-SACKs, the throughput for an unordered data transfer over SCTP is improved by ~14% for a data sender with 32KB send buffer under low (~0-1%) loss rate.

1.2.3 Problem Statement

Suppose that reliable transport protocols were designed to NOT tolerate reneging. What would be the advantages and disadvantages? In such a case, the send buffer utilization would be always optimal, and the application throughput would be improved for data transfers with constrained send buffers. Current transport protocols employing SACKs such as TCP and SCTP suffer because of the assumption that reneging may happen. Note that, a non-reneging transport protocol (that is when all out-of-order data are non-renegable) would perform even better than a protocol using NR-SACKs since there is no constraint on data delivery service used.

If we can document that reneging never happens or happens rarely, we can argue that reliable transport protocols should be modified to operate on the assumption that all data are non-renegable. As simplified argument, let us assume that reneging happens rarely, say once in a million TCP flows. Case A (current practice): TCP implementations tolerate reneging to maintain the reliable data transfer of the single reneging connection. The 999999 non-reneging connections waste ~10% of the main memory allocated for send buffer (under mild (~1-2%) loss rates) and achieve lower application throughput.

Case B (proposed change): TCP does not tolerate reneging. For our simplified argument, 999999 connections have improved performance and 1 connection gets RESET.

Changing TCP or SCTP with their current support for reneging into non-reneging transport protocols requires only minor modifications to current practice. First, the semantics for SACK is changed from *advisory* to *permanent*. Once a data receiver SACKs data, that out-of-order data may not be reneged (Note: with this simple change, the NR-SACK extension is not needed; SACKed data become non-renegable.) If a data receiver does have to recapture allocated receive buffer space, we propose that the data receiver MUST RESET the connection prior to reneging (i.e., only penalize the reneging connection). A data sender needs no mechanism to handle reneging, since the data receiver must reset the connection when reneging is necessary.

Our hypothesis is that the inefficiency of the transport protocols due to possible reneging needs to be corrected by designing TCP and SCTP to not tolerate reneging. We argue that penalizing a few reneging connections by making them RESET worthwhile so that the large majority of non-reneging connections benefit from better send buffer utilization and increased throughput.

One might criticize our proposed change, "What if a data receiver were to renege on SACKed data and not RESET?" That incorrect behavior would cause a failure in the reliable data transfer or a deadlock to occur. But currently a TCP receiver may not renege on cumulatively ACKed data. If a TCP data receiver did such an incorrect behavior, a failure or deadlock will occur. Our proposed change simply defines SACKed data to have the same status as ACKed data.

Our original expectation was that reneging never happens in practice. As it turns out, this dissertation research observes that data reneging does happen, albeit rarely. Given this observation, we characterize the circumstances causing reneging. Once the circumstances are known, we analyze the pros and cons of reneging on the operating system's operation. If reneging does not help the operating system to resume its operation, reneging should be disabled. Randall Stewart, the designer of SCTP and implementor of the its FreeBSD reference implementation, hypothesized that an operating system would eventually crash if the operating system ever arrived to a state in which reneging was needed. If so, then why bother tolerating data reneging!

To better understand reneging in current practice, this dissertation identifies operating systems that have built-in mechanisms for reneging, and ones that do not (Chapter 4). If the majority of the operating systems did not have mechanisms to deal with reneging, employing the current SACK mechanism would be inefficient and designing non-reneging transport protocols would be absolutely called for. Our investigation reveals that several operating systems (FreeBSD, Linux, Mac OS, Solaris, and Windows) have reneging mechanisms.

Simply put – does reneging occur or not? We know of only one study of reneging (an MS thesis) in the research community. We do not know what percentage of connections renege, nor if today's TCP implementations handle reneging instances correctly. The Internet is evolving continuously; we should model actual practice. By analogy, a study by [Medina 2005] examined a large number of web-servers and showed that Tahoe TCP was used in only 2.6% of web-servers in 2005. Thus, there is no need to compare new TCP extensions with TCP Tahoe since TCP Tahoe is now past practice. If we observe reneging occurs rarely or never, we will have evidence to change the basic assumptions of transport layer protocols.

1.2.4 Research Goals – Why Study Data Reneging?

The primary research goal is to investigate reneging in the current Internet and attempt to detect reneging instances through a passive measurement technique. Findings from the passive measurement analysis of Internet traces are presented in Chapter 3.

A secondary research goal is to design a tool to cause a remote machine to renege. Inspecting reneging mechanisms in various operating systems (Chapter 4) provides a basis for building a reneging causing tool. Chapter 5 presents a tool to cause a machine to renege, and investigates the effects of reneging on transport connections and operating systems.

1.3 Related Research

To the best of this author's knowledge, the first and only prior study of reneging is [Blanton 2008]. In this MS thesis which was not published elsewhere, 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.

In general, a TCP data sender is not designed to detect reneging. Instead, a TCP sender is designed to tolerate reneging as specified in [RFC2018]. The SACK scoreboard should be cleared at a retransmission timeout (RTO) and the segment at the left edge of the window must be retransmitted. In [Blanton 2008], the author hypothesized that discarding the SACK scoreboard 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 SACK scoreboard when reneging is detected instead of waiting until the RTO. The reneging detection algorithm compares existing SACK blocks (scoreboard) with incoming ACKs and when an ACK is advanced to the middle of a SACK block, reneging is detected. Using real traces, the author analyzed TCP connections with SACKs to report frequency of reneging. Out of 1,306,646 connections analyzed, the author's reneging detection algorithm identified 227 connections (0.017%) having reneged. The author concluded that reneging is an infrequent event, and in general was found in systems running servers on well-known ports (email servers, HTTP servers.) Another finding is that multiple instances of reneging were often observed in a single connection.

The reneging detection algorithm proposed in [Blanton 2008] is simple, robust to packet reordering, and does not rely on any external timers or events. The algorithm does not detect reneging until an ACK advances to the middle of a SACK block. The author acknowledges that reneging can be detected earlier when the TCP receiver skips previously SACKed data. For such a case, SACKs are used for reneging detection. The author is concerned that reordered ACKs would look like reneging with this technique, so a mechanism is needed to ensure that ACKs are not reordered. For that, the author suggests the use of TCP timestamps [RFC1323]. Unfortunately, ACKs from the same window in general have the same TCP timestamp value which makes timestamps less robust to reordering check. Our approach to detect reneging, detailed in Section 3.2, uses both ACKs and SACKs. To infer ACK reordering, our approach uses IP ID and TCP ACK fields instead of TCP timestamps.

During the trace analysis [Blanton 2008], a common behavior is observed. An ACK would advance to the middle of a SACK block and the next ACK observed within 5ms would cover the entire SACK block. That type of reneging is referred as “spurious” reneging. Our approach to detect reneging relies on retransmissions and ignores “spurious” reneging instances when there are no retransmissions.

[Paxson 1997] presents “*tcpanaly*” a tool which automatically analyses the correctness of TCP implementations by inspecting passive traces collected for bulk data transfers in both directions (data and ACK traffic). The tool can identify large number of TCP implementations employed at the time and reports errors when the TCP flows inspected show non-conformant TCP behavior. In [Paxson 1997], the main interests are data sender’s congestion window evolution and data receiver’s proper ACK generation. With analysis, non-conformant TCP stacks are identified and reported to the stack implementors. Similar to the [Paxson 1997], we detect reneging instances through a passive measurement as detailed in Section 3.2 using bidirectional TCP traffic.

[Padhye 2001] describes the TCP Behavior Inference Tool (TBIT) [Tbit] which is used to infer the TCP behavior of remote web servers. The authors define a number of test cases that can determine, for examples, the initial congestion window size, the congestion control algorithm used, the time wait duration time, and ECN usage of web servers.

The test case of importance to this research is the SACK mechanism test. This test checks if a web server supports SACKs. A TCP end-point acknowledges its peer that it is SACK enabled by sending a SACK-Permitted option in the SYN/SYN-ACK packet [RFC2018]. If the web server is SACK enabled, this test further checks if

SACKs sent by TBIT are correctly processed by the web server when it retransmits segments during the loss recovery period.

In 2001, out of 4550 web servers tested by TBIT, only 1854 (~41%) were SACK enabled. The authors also reported that only 42% of SACK enabled web servers used SACK information correctly, the rest did not use SACK information to minimize retransmissions during loss recovery.

Our tool to cause a remote machine to renege (see Section 5.1) is based on TBIT, and needs to send specific sequences of TCP PDUs. The TCP traffic generated by TBIT is restricted not to be hostile to the remote web servers. On the other hand, our reneging causing tool tries to exhaust a remote machine's main memory as much as possible to trigger a reneging instance. TCP PDUs generated by our tool are hostile and may eventually cause the remote machine to renege or even crash.

[Fraleigh 2003] describes the architecture and capabilities of the IPMON system which is used for IP monitoring at Sprint IP backbone network. IPMON consists of passive monitoring entities, a data repository to store collected trace files and an offline analysis platform to analyze the collected data. The authors analyze individual flows and traffic generated by different protocols and applications. The authors present statistics such as traffic load (weekly and daily), traffic load by applications (web, mail, file transfer, p2p, streaming), traffic load in flows. Also TCP related statistics such as packet size distribution, RTT, out-of-sequence rate, and delay distributions are presented. IPMON is another passive measurement tool as is tcpanaly [Paxson 1997], and our method for detecting reneging instances (presented in Section 3.2).

In [Jaiswal 2004], the authors introduce a passive measurement technique to infer and keep track of the congestion window (cwnd) and round trip time (RTT) of a TCP data sender. To infer a data sender' cwnd, the authors construct a replica of the data sender's TCP state using a finite state machine (FSM). The FSM is updated through ACKs and retransmissions seen at the data collection point. We employ the same technique to update the view of a data receiver's receive buffer. This view is then used to detect reneging instances (detailed in Chapter 3.)

Using passive monitoring at an intermediate point, the cwnd evolution may be under-estimated when three dup ACKs get lost after the intermediate point or over-estimated if an entire window of packets gets lost before reaching to the intermediate point. Our passive measurement approach to detect reneging is more robust to SACK losses when compared to [Jaiswal 2004]. The robustness comes from the SACKs being cumulative. Some of the information contained in lost or missing SACK segments will be learnt from subsequent SACKs.

[Medina 2004] investigates the effect of "middleboxes" (firewalls, NATs, proxies, etc.) on the performance of IP and TCP protocols. The authors use TBIT, the tool described in [Padhye 2001], to send SYN packets with various IP or TCP options to web servers in order to detect how middleboxes react to the options.

When no IP options are sent, most of the connections (98%) are established. When IP options such as Record Route, Timestamp, and Unallocated Option are present, the number of established connections drops dramatically to 45%, 36% and 0%, respectively. On the other hand, middleboxes have little effect on connection establishment (connection failures are around 3%) when TCP options such as Timestamp and Unallocated Option are present.

[Ladha 2004] is an independent and parallel work to [Medina 2005] in which the authors measure the current deployment status of recent TCP enhancements using the TBIT tool. The authors added three new tests cases for recent TCP extensions (limited transmit, appropriate byte counting (ABC), and early retransmit) to the TBIT. In addition to the new test cases added, the SACK and initial congestion window tests of [Padhye 2001] are rerun to evaluate the deployment status of these extensions. A simulation study is performed to evaluate the performance of TCP extensions mentioned above against TCP New Reno. TCP New Reno is compared against TCP SACK with each extension added one at a time based on the standardization time in the IETF.

In [Medina 2005], the authors investigate the correctness of modern TCP implementations through active and passive measurements. The active measurements are taken by the use of TBIT tool. All TBIT tests are rerun and compared with 2001's results [Padhye 2001]. Also, new test cases such as Byte Counting and Limited Transmit are added to TBIT where the deployment status of new extensions to TCP is explored.

SACK related results are of particular interest. SACK-enabled web servers increased from being 41% in 2001 to 68% in 2004. Half of the SACK enabled web servers also implement D-SACK [RFC2883]. Also the correct use of SACK information by data senders (web servers) increased more dramatically: 90% in 2004 as opposed to 2001's 42%. A new test is introduced to test if the web servers correctly generate SACK blocks and around 91% of the web servers tested generated correct SACK blocks.

The authors also extended their previous work [Medina 2004] through passive packet trace analysis to characterize the TCP behavior of the data receivers (web clients). Statistics for advertised window and TCP options such as window scale factor, timestamp, ECN capability and advertised MSS are provided for web clients. The authors suggest developing tools to validate new transport protocols such as SCTP and Datagram Congestion Control Protocol (DCCP).

Three studies summarized above, [Medina 2004], [Ladha 2004] and [Medina 2005], are active measurement studies as is our tool to cause a remote host to renege. The main difference between these studies and our proposed study is the amount of data traffic generated. In general, studies above send a small number of TCP PDUs using a single TCP connection. Our reneging tool, on the other hand, requires sending large amount of out-of-order data in order to cause reneging by consuming the main memory allocated for network buffers using parallel TCP connections. While the traffic generated by above studies is harmless to the tested web server, our tool to cause a remote host to renege may cause the remote host to renege or even crash.

Chapter 2

MISBEHAVIORS IN TCP SELECTIVE ACKNOWLEDGMENT (SACK) GENERATION

While analyzing Internet traces of TCP traffic to detect instances of data reneging, detailed in Chapter 3, we frequently observed seven misbehaviors in the generation of TCP SACKs. These misbehaviors gave us the impression that data reneging was happening frequently. Upon closer inspection of the reneging instances, we concluded that in fact some TCP implementations were generating SACKs incompletely under some circumstances. To confirm whether or not the misbehaviors observed in the Internet traces were actual reneging instances (misbehaving TCP stacks), we tested the RFC 2018 conformant SACK generation on wide range of operating systems. In our testing, we simply mimicked the traffic behavior observed in the Internet traces prior to observed misbehaviors.

In this chapter, we present a methodology and its application to test a wide range of operating systems for SACK generation. The research findings for this chapter appear in the journal paper [Ekiz 2011b].

2.1 Introduction

The Selective Acknowledgment (SACK) mechanism, [RFC2018], an extension to Transmission Control Protocol's (TCP) [RFC793] ACK mechanism, allows a data receiver to explicitly acknowledge arrived *out-of-order* data to a data sender. When using SACKs, a TCP data sender need not retransmit SACKed data during the loss recovery period. Previous research [Allman 1997], [Bruyeron 1998], [Fall 1996] showed that SACKs improve TCP throughput when multiple losses occur within the same window. The success of a SACK-based loss recovery algorithm [RFC3517] is proportional to the SACK information received from the data receiver. In this research, we investigate RFC 2018 conformant SACK generation.

Deployment of the SACK option in TCP connections has been a slow, but steadily increasing trend. In 2001, 41% of the web servers tested were SACK-enabled [Padhye 2001]. In 2004, SACK-enabled web servers increased to 68% [Medina 2005]. All of the operating systems tested in this study accept SACK-permitted TCP connections.

Today's reliable transport protocols such as TCP [RFC793] and SCTP [RFC4960] are designed to tolerate data receiver reneging (simply, data reneging) (Section 8 of [RFC2018]). As defined in Section 1.2.1, data reneging occurs when a data receiver SACKs data, and later discards that data from its receive buffer prior to delivering it to a receiving application (or receiving socket buffer).

In our research, we argue that reliable transport protocols should not be designed to tolerate data reneging, largely because we found data reneging occurs rarely in practice. While developing our software to discover data reneging in trace data, Section 3.2 in Chapter 3, we analyzed TCP SACK information within Internet traces provided by the Cooperative Association for Internet Data Analysis (CAIDA) [Caida]. At first it seemed that data reneging was happening frequently contrary to our hypothesis. On closer inspection however, it appeared that the generation of SACKs in many TCP connections potentially was incorrect according to RFC 2018. Sometimes SACK information that should have been sent was not. Sometimes the wrong SACK information was sent. In one misbehavior, SACKs from one connection were sent in the

SYN-ACK used to open a later connection! These misbehaviors wrongly gave the impression that data reneging was occurring.

Our discovery led us to verifying SACK generation behavior of TCP data receivers for a wide range of operating systems. In our research, our goal is to present a methodology for verifying SACK behavior, and to apply the methodology to report misbehaving TCP stacks. The goal of the research is not to measure how much the misbehaviors degrade the performance, but rather to identify misbehaving TCP stacks so they will be corrected.

We first present in Section 2.2 seven misbehaviors, five (A-E) observed in the CAIDA traces, and two (F-G) additional SACK related misbehaviors observed during our testing of A-E. Technically, misbehaviors A-E indicate that SHOULD requirements of [RFC2018] are not being followed, and SHOULD means "that there may exist valid reasons in particular circumstances to ignore a particular item, but the full implications must be understood and carefully weighed before choosing a different course" [RFC2119]. Upon analysis, we believe these misbehaviors to be accidental, not incidental.

Misbehaviors A-F can reduce the effectiveness of SACKs. Misbehavior G is the worst one where a data receiver transmits a SACK for data that was never received, thus questioning the data transfer reliability of the connection. To discover which implementations are misbehaving, we defined seven test extensions to the TCP Behavior Inference Tool (TBIT) [Tbit], a tool that verifies TCP endpoint behavior.

The methodology using TBIT is described in Section 2.3, and the results of our TBIT tests are presented in Section 2.4. Section 2.5 summarizes our research in SACK generation misbehavior.

2.2 Testing Seven SACK Misbehaviors

The five SACK generation misbehaviors observed in CAIDA traces are described as:

A. Fewer than max number of reported SACKs

B. Receiving data between CumACK and first SACK

C. Receiving data between two previous SACKs

D. Failure to report SACKs in FIN segments

E. Failure to report SACKs during bidirectional data flow

The two additional SACK-related misbehaviors observed during our TBIT testing of A-E are:

F. Mishandling of data due to SACK processing

G. SACK reappearance in consecutive connections

2.2.1 Fewer than Max Number of Reported SACKs

RFC 2018 Section 3 specifies that "*the data receiver SHOULD include as many distinct SACK blocks possible in the SACK option*," and that "*the 40 bytes available for TCP options can specify a maximum of four SACK blocks.*" For some TCP flows, we observed that only two or sometimes three SACK blocks were reported by a data receiver even though more SACKs were available and additional space existed in the TCP header.

That is, more than two SACK blocks at the data receiver are known to exist (say X_l-X_r, Y_l-Y_r, and Z_l-Z_r) but only two SACK blocks are reported (X_l-X_r and Y_l-Y_r). A SACK block is presented with the following notation: X_l-X_r, where X_l and X_r stand for the left and right edge, respectively. When the cumulative ACK advances beyond X_r, SACK block X_l-X_r, is correctly no longer reported, and SACK block Z_l-Z_r is reported along with block Y_l-Y_r. This misbehavior implies that the data receiver reports less than the recommended maximum SACK blocks.

We extended the existing TBIT test "SackRcvr" [Tbit] to determine a receiver's maximum number of reported SACK blocks. For clarity, most TCP segments sent by TBIT in our Figures 2.1-2.7 are shown to carry 1 byte of data and create 1 byte gaps. This numbering scheme makes the TBIT tests easy to understand. In the actual tests performed (see traces [Ekiz 2011c]), segments carry 1460 bytes of data and create 1460 byte gaps. The only exception was for Tests A, F for Linux systems. The Linux advertised receiver window is only 5840 bytes. To simulate 4 gaps, TBIT segments for two Linux tests carry 600 bytes of data and create 600 byte gaps.

The TBIT test in Figure 2.1 operates as follows. Sequence numbers of segments are shown in parenthesis:

Test A

1. TBIT establishes a connection to TCP Implementation Under Test (IUT) with SACK-Permitted option and Initial Sequence Number (ISN) 400
2. IUT replies with SACK-Permitted option
3. TBIT sends segment (401) in order
4. IUT acks the in order data with ACK (402)
5. TBIT sends segment (403) creating a gap at IUT
6. IUT acks the out-of-order data with SACK
7. TBIT sends segment (405) creating 2nd gap at IUT
8. IUT acks the out-of-order data with SACK

9. TBIT sends segment (407) creating 3rd gap at IUT

10. IUT acks the out-of-order data with SACK

11. TBIT sends segment (409) creating 4th gap at IUT

12. IUT acks the out-of-order data with SACK

13. TBIT sends three resets (RST) to abort the connection

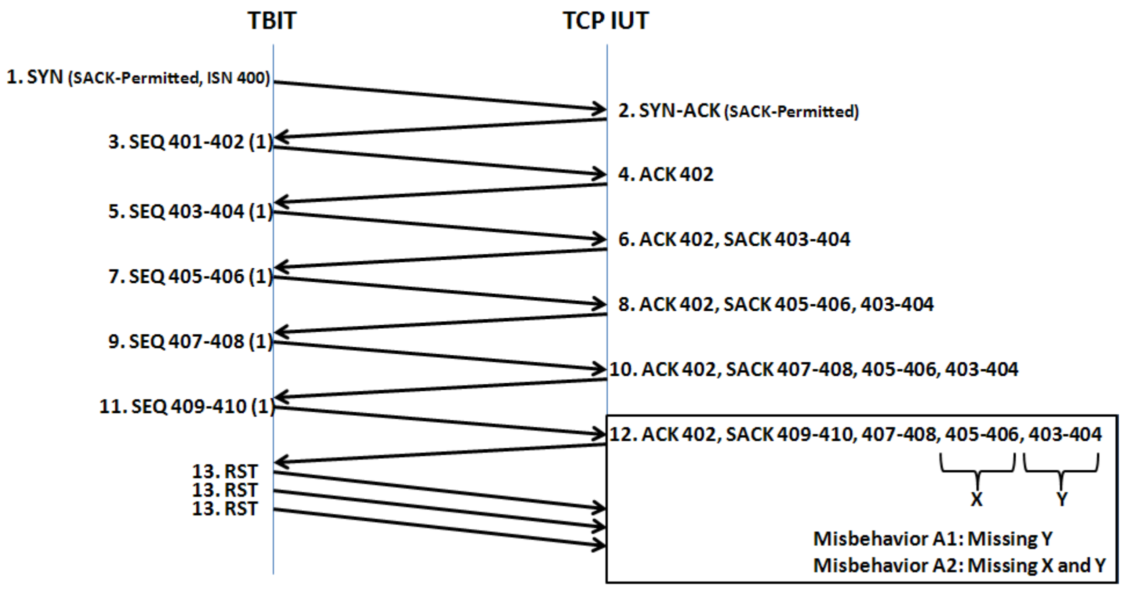

TBIT
TCP IUT
1. SYN (SACK-Permitted, ISN 400)
2. SYN-ACK (SACK-Permitted)
3. SEQ 401-402 (1)
4. ACK 402
5. SEQ 403-404 (1)
6. ACK 402, SACK 403-404
7. SEQ 405-406 (1)
8. ACK 402, SACK 405-406, 403-404
9. SEQ 407-408 (1)
10. ACK 402, SACK 407-408, 405-406, 403-404
11. SEQ 409-410 (1)
12. ACK 402, SACK 409-410, 407-408, 405-406, 403-404
X
Y
13. RST
13. RST
13. RST
Misbehavior A1: Missing Y
Misbehavior A2: Missing X and Y

Figure 2.1: Fewer than max number of reported SACKs

The last SACK from the IUT reflects an implementation's support for maximum number of SACK blocks reported. A conformant implementation's last SACK should be as SACK #12 in Figure 2.1. A misbehaving implementation would not SACK block Y (Misbehavior A1), or blocks X and Y (Misbehavior A2).

2.2.2 Receiving Data between CumACK and First SACK

For some TCP flows having at least two SACK blocks, we observed the following misbehavior. Once the data between the cumulative ACK and the first SACK block was received, the data receiver increased the cumulative ACK, but misbehaved and did not acknowledge other SACK blocks. (The acknowledgment with no SACK blocks implies an instance of data reneging.)

RFC 2018 specifies that: "*If sent at all, SACK options SHOULD be included in all ACKs which do not ACK the highest sequence number in the data receiver's queue*." So, SACKs should be included when the cumulative ACK is increased and out-of-order data exists in the receive buffer.

Test B, illustrated in Figure 2.2, checks this misbehavior. The second SACK block should remain present when the cumulative ACK is increased beyond the first SACK block but is less than the second SACK block.

Test B

1. TBIT establishes a connection to IUT with SACK-Permitted option and ISN 400
2. IUT replies with SACK-Permitted option
3. TBIT sends segment (401) in order
4. IUT acks the in order data with ACK (402)
5. TBIT sends segment (404) creating a gap at IUT (the gap between Cum ACK and first SACK block)
6. IUT acks the out-of-order data with SACK
7. TBIT sends segment (406) creating 2nd gap at IUT
8. IUT acks the out-of-order data with SACK
9. TBIT sends segment (403)

10. IUT acks the out-of-order data with SACK

11. TBIT sends segment (402) to fill the gap between Cum ACK and first SACK

12. IUT acks the in order data with SACK

13. TBIT sends three RSTs to abort the connection

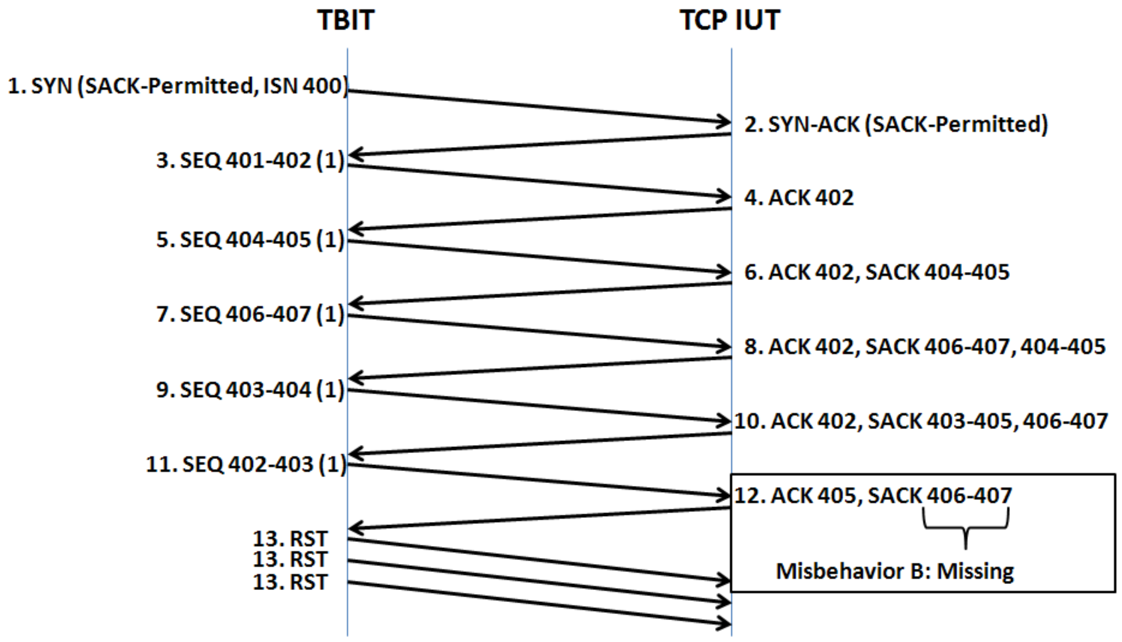


Figure 2.2: Receiving data between CumACK and first SACK

A conformant implementation should report SACK block (406-407) as shown in #12 in Figure 2.2. A misbehaving implementation omits reporting the SACK block.

2.2.3 Receiving Data between Two Previous SACKs

We observed that some TCP flows report SACK information incompletely once the missing data between two SACK blocks (say X_l-X_r and Y_l-Y_r) are received. The next

SACK should report a single SACK block concatenating the first SACK block (X_l-X_r), the missing data in between, and the second SACK block (Y_l-Y_r). Instead some implementations generate a SACK covering only the first SACK block and the missing data, i.e., (X_l-Y_l), omitting the second SACK block. This behavior implies that the second SACK block is reneged.

Test C, illustrated in Figure 2.3, tests this misbehavior. The data receiver should report one SACK block covering the two SACK blocks and the data in between.

Test C

1. TBIT establishes a connection to IUT with SACK-Permitted option and ISN 400
2. IUT replies with SACK-Permitted option
3. TBIT sends segment (401) in order
4. IUT acks the in order data with ACK (402)
5. TBIT sends segment (403) creating a gap at IUT
6. IUT acks the out-of-order data with SACK
7. TBIT sends segment (405) creating 2nd gap at IUT
8. IUT acks the out-of-order data with SACK
9. TBIT sends segment (404) with missing data between the first and the second SACK blocks
10. IUT acks the out-of-order data with SACK
11. TBIT sends three RSTs to abort the connection

A proper implementation is expected to report the out-of-order data (403-406) as shown in #10 in Figure 2.3. A misbehaving implementation would report the SACK block partially (403-405).

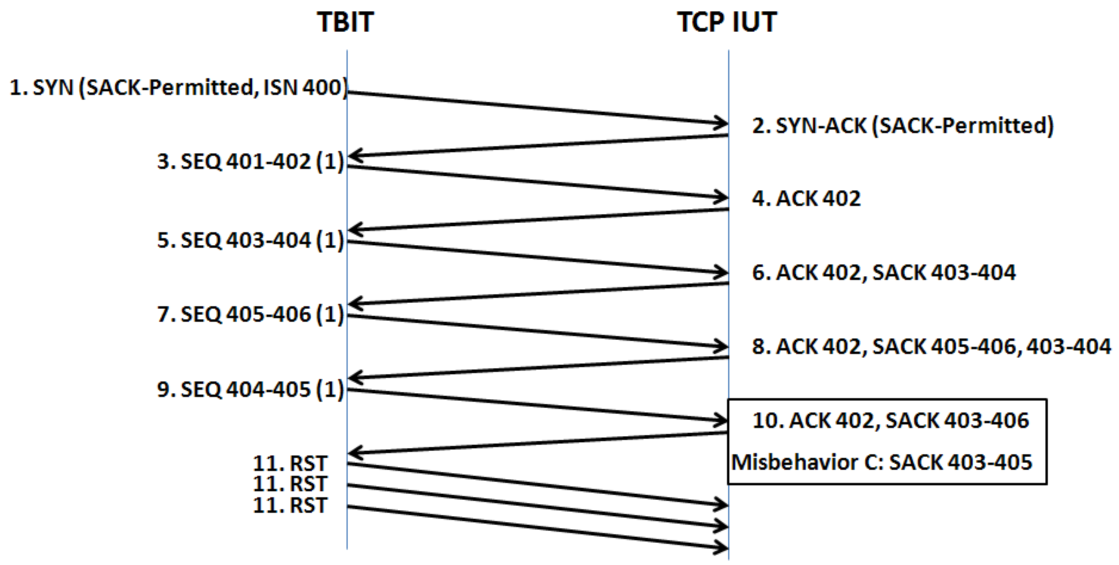

TBIT
TCP IUT
1. SYN (SACK-Permitted, ISN 400)
2. SYN-ACK (SACK-Permitted)
3. SEQ 401-402 (1)
4. ACK 402
5. SEQ 403-404 (1)
6. ACK 402, SACK 403-404
7. SEQ 405-406 (1)
8. ACK 402, SACK 405-406, 403-404
9. SEQ 404-405 (1)
10. ACK 402, SACK 403-406
Misbehavior C: SACK 403-405
11. RST
11. RST
11. RST

Figure 2.3: Receiving data between two previous SACKs

2.2.4 Failure to Report SACKs in FIN Segments

When closing a connection, a receiving side sends a FIN segment along with the acknowledgment (ACK and SACK) for the data received. But for some data flows, we observed the FIN segment does not carry SACK information. As discussed in Section 2.2.2, the receiver should include the SACK information along with the ACK.

Test D, in Figure 2.4, operates as follows: TBIT opens a connection and sends a GET request (HTTP/1.0) to the IUT. The IUT sends the requested data, and immediately closes the connection with a FIN since HTTP/1.0 is non-persistent [RFC1945].

Test D

1. TBIT establishes a connection to IUT with SACK-Permitted option and ISN 400
2. IUT replies with SACK-Permitted option
3. TBIT sends segment (401-450: GET /index.pdf HTTP/1.0 request) in order

4. IUT acks the in order data with ACK (450)
5. IUT starts sending segments with contents of index.pdf
6. TBIT sends segment (451) creating a gap at IUT
7. TBIT acks segments of IUT
8. IUT acks the out-of-order data with SACK
9. IUT continues sending contents of index.pdf with SACK
10. Once index.pdf is sent completely, IUT sends a FIN to close the connection

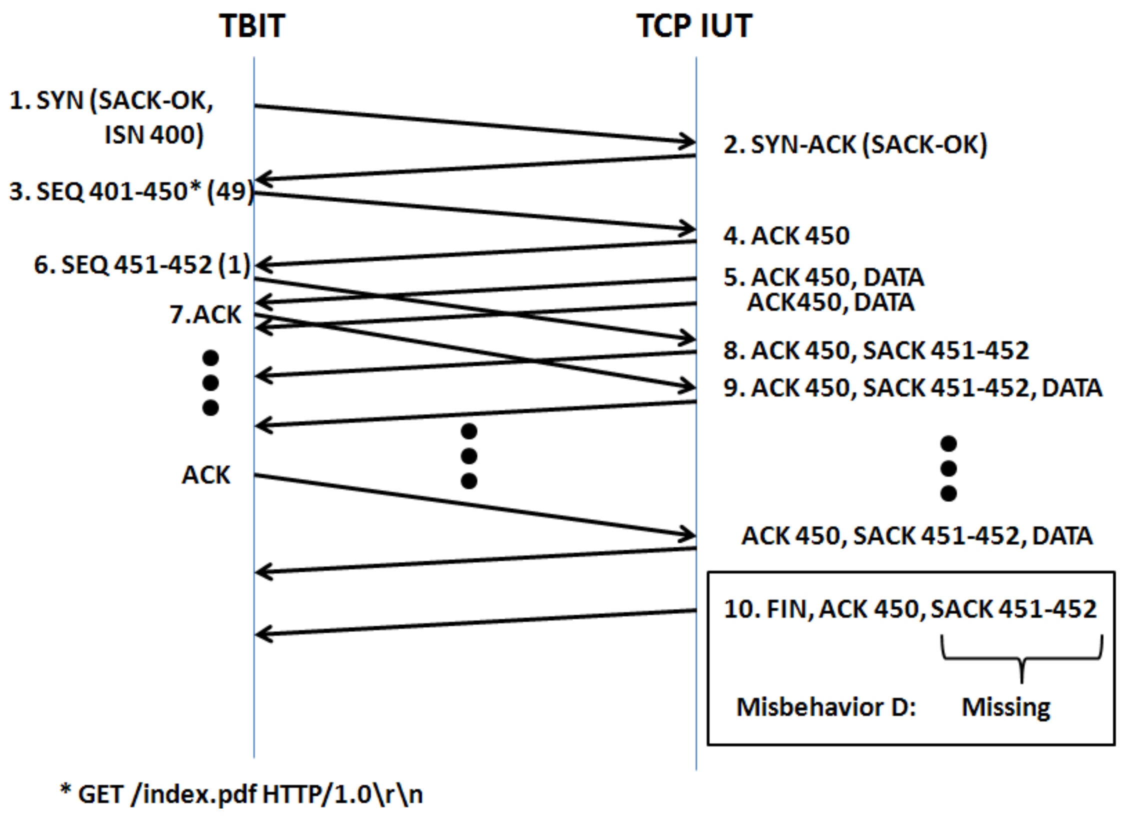


Figure 2.4: Failure to report SACKs in FIN segments

The conformed behavior of a data receiver is to include SACK information in the FIN segment as shown in #10 in Figure 2.4. A misbehaving implementation sends an ACK, but no SACK information.

2.2.5 Failure to Report SACKs during Bidirectional Data Flow

This misbehavior occurs when the data flow is bidirectional. In some TCP flows, SACK information is not conveyed when the TCP segment carries data. If a TCP host is sending data continuously (e.g., an HTTP server), only one SACK is sent when out-of-order data are received, and SACK information is not piggybacked with the following segments. This misbehavior can cause less efficient SACK-based loss recovery since SACKs are sent only once for each out-of-order data arrival.

As stated in Section 2.2.2, a conformant data receiver should include SACK information with all ACKs. If ACKs are piggybacked while sending data, SACKs should also be piggybacked in the TCP segments.

We added a new TBIT test for misbehavior E. To have bidirectional data flow and out-of-order data simultaneously, we used HTTP/1.1 GET requests [RFC2616]. HTTP/1.1 opens a persistent connection between TBIT and an IUT. TBIT requests the file index.pdf (11650 bytes) which is large enough to have a data transfer requiring several round trips so that SACK information can be observed in the segments.

Test E

1. TBIT establishes a connection to IUT with SACK-Permitted option and ISN 400
2. IUT replies with SACK-Permitted option
3. TBIT sends segment (401-450: GET /index.pdf HTTP/1.1 request) in order
4. IUT acks the in order data with ACK (450)

5. IUT starts sending segments with contents of index.pdf
6. TBIT sends segment (451) creating a gap at IUT
7. TBIT acks segments of IUT
8. IUT acks the out-of-order data with SACK
9. IUT continues sending contents of index.pdf with SACK
10. Once index.pdf is retrieved completely, TBIT sends three RSTs to abort the persistent connection

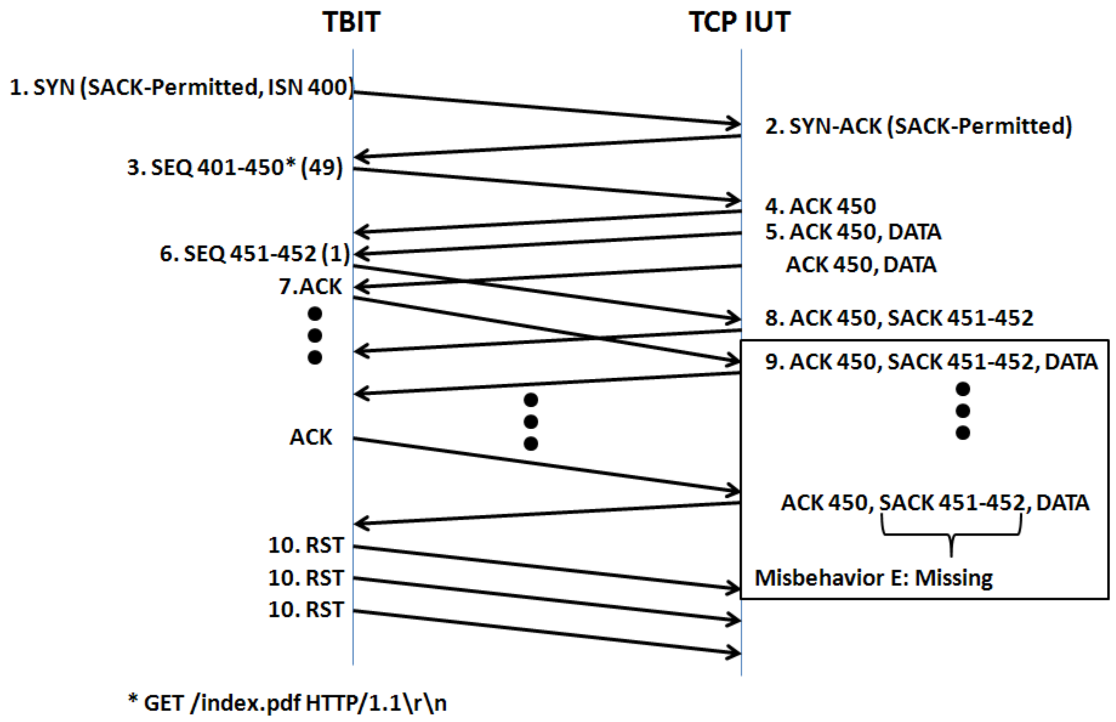


Figure 2.5: Failure to report SACKs during bidirectional data

A conformant implementation appends SACK information in TCP segments carrying data as shown in Figure 2.5, whereas a misbehaving implementation does not.

2.2.6 Mishandling of Data Due to SACK Processing

While running Test E, we observed another SACK-related misbehavior. Some segments do not carry maximal payload when SACKs are included. Rather they carry only the number of bytes equal to the SACK information appended.

We explain the misbehavior in detail using Test F shown in Figure 2.6. Test F modifies Test E. Instead of sending one out-of-order data, four are sent to check how data is sent by the TCP IUT as the number of appended SACKs increases.

Test F

1-5. Same as Test E

6. TBIT sends segment (451) creating a gap at IUT, and ACKing the 1st segment of IUT
7. When the ACK for 1st segment of IUT is received, IUT's congestion window (cwnd) is increased enabling sending two new segments. IUT sends two segments with one SACK block: 3rd segment (1448 bytes) and 4th segment (12 bytes)
8. TBIT sends segment (453) creating a second gap at IUT, and ACKing the 2nd segment of IUT
9. When the ACK for 2nd segment of IUT is received, IUT sends two segments each with two SACKs: 5th segment (1440 bytes) and 6th segment (20 bytes)
10. TBIT sends segment (455) creating a third gap at IUT, and ACKing the 3rd segment of IUT
11. When the ACK for 3rd segment of IUT is received, IUT sends two segments each with three SACKs: 7th segment (1432 bytes) and 8th segment (28 bytes)
12. TBIT sends segment (457) creating a fourth gap at IUT, and ACKing the 4th segment of IUT
13. When the ACK for 4th segment of IUT is received, IUT sends two segments each with four SACKs: 9th segment (1424 bytes) and 10th segment (36 bytes)

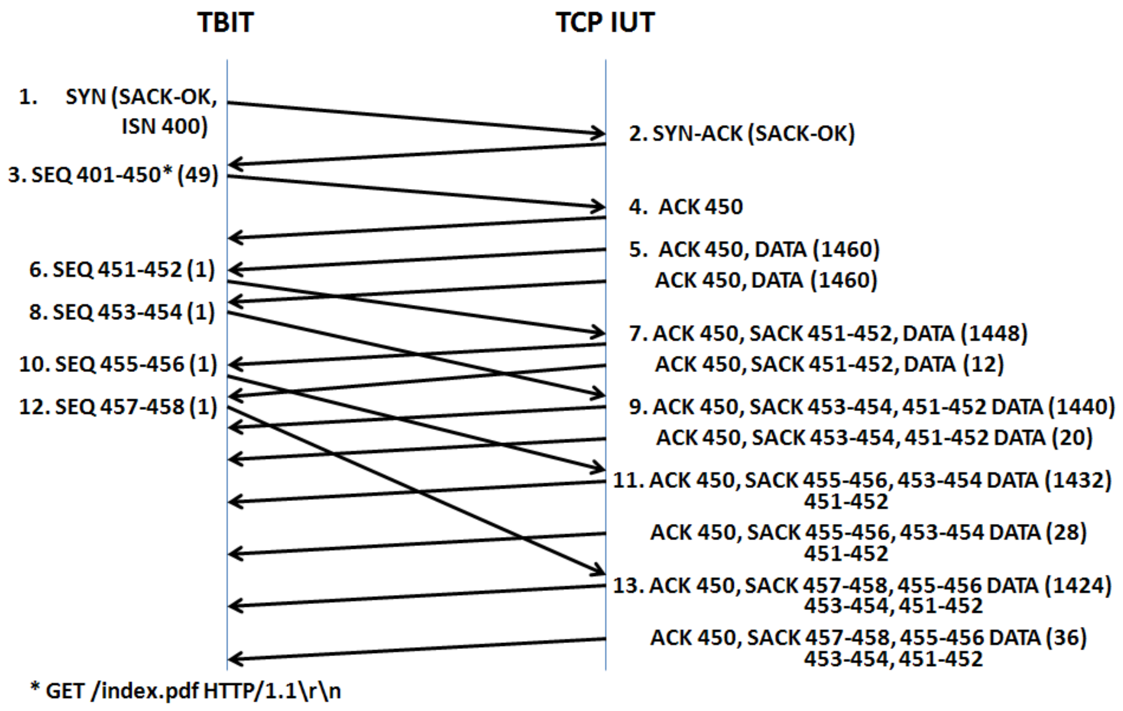

TBIT
TCP IUT
1. SYN (SACK-OK, ISN 400)
2. SYN-ACK (SACK-OK)
3. SEQ 401-450* (49)
4. ACK 450
5. ACK 450, DATA (1460)
ACK 450, DATA (1460)
6. SEQ 451-452 (1)
8. SEQ 453-454 (1)
7. ACK 450, SACK 451-452, DATA (1448)
ACK 450, SACK 451-452, DATA (12)
10. SEQ 455-456 (1)
12. SEQ 457-458 (1)
9. ACK 450, SACK 453-454, 451-452 DATA (1440)
ACK 450, SACK 453-454, 451-452 DATA (20)
11. ACK 450, SACK 455-456, 453-454 DATA (1432) 451-452
ACK 450, SACK 455-456, 453-454 DATA (28) 451-452
13. ACK 450, SACK 457-458, 455-456 DATA (1424) 453-454, 451-452
ACK 450, SACK 457-458, 455-456 DATA (36) 453-454, 451-452
* GET /index.pdf HTTP/1.1\r\n

Figure 2.6: Mishandling of data due to SACK processing

For every ACK received from TBIT, the IUT's cwnd is increased to send two new segments. After the first ACK is received, the IUT sends segments with 1448 and 12(!) bytes of data, respectively. Both segments from the IUT do include a SACK block. A proper SACK implementation is expected to send 1448 bytes of data in both segments each with 12 bytes of SACK in the TCP options. As the number of SACKs increase to 2, 3 and 4, the IUT sends two segments with (1440, 20), (1432, 28), (1424, 36) bytes, respectively. Note that the second segment always (coincidentally?) carries a number of data bytes equal to bytes needed for the SACK blocks, not a full size segment. This misbehavior is observed continuously while out-of-order data exists at the IUT. Throughput is decreased almost in half for the time when out-of-order data exists in the receive buffer.

2.2.7 SACK Reappearance in Consecutive Connections

When verifying misbehaviors A-E, we ran the TBIT tests successively using different port numbers. We observed that in some TCP stacks, SACK information of a prior connection, say from Test A, would sometimes appear in the SYN-ACK segment of a new connection, say from Test B!

To further investigate the misbehavior, we developed Test G as shown in Figure 2.7. This test purposely uses the same initial sequence numbers for consecutive connections to demonstrate a worst case:

Test G

1. TBIT establishes a connection to IUT with SACK-Permitted option and ISN 400 on ephemeral port Eph_1
2. IUT replies with SACK-Permitted option on port 80
3. TBIT sends segment (401) in order
4. IUT acks the in order data with ACK (402)
5. TBIT sends segment (403) creating a gap at IUT
6. IUT acks the out-of-order data with SACK
7. TBIT sends three RSTs segments to abort the connection
8. After ‘X’ minutes, TBIT establishes a connection to IUT with SACK-Permitted option and ISN 400 on ephemeral port Eph_2
9. IUT replies with SACK-Permitted option on port 80 *including a SACK block of the previous connection*

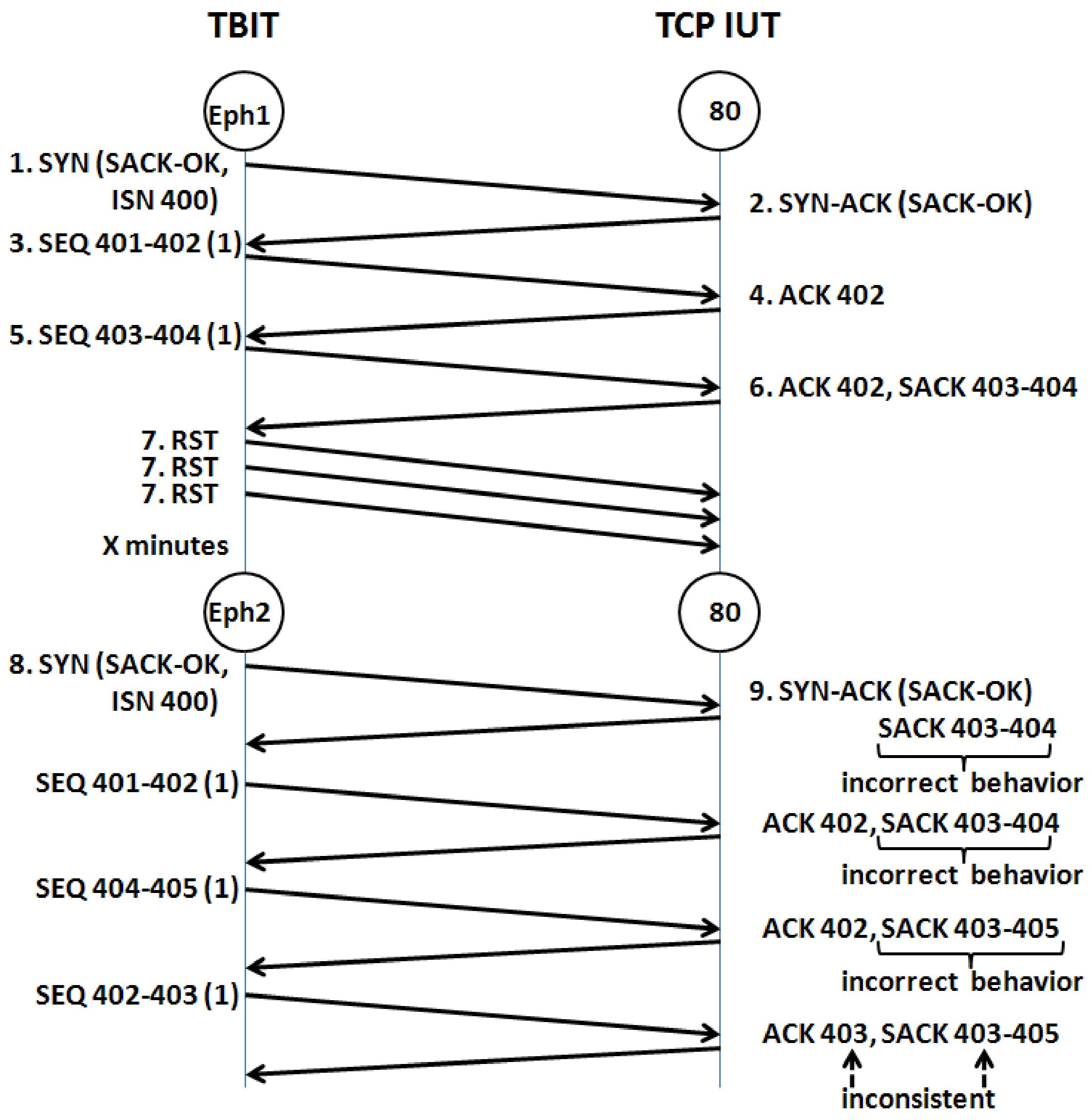


Figure 2.7: SACK reappearance in consecutive connections

In the second connection, the IUT sends an acknowledgment with SACK block 403-404 which is from the first connection. TBIT assumes 403 is SACKed, but the IUT never received the data. TBIT later sends data 402-403 to check if the IUT increases ACK to 405. The IUT returns an inconsistent ACK 403, SACK 403-405, but fortunately does not increase ACK to 405 so the connection remains reliable. In a real connection, eventually the sender will timeout on 403, discard all SACKed information, and retransmit the data, thus returning to a correct state [RFC2018]. However for a brief period of time, the data sender and receiver are in an inconsistent state.

2.3 Experimental Design

The TBIT tests described in Section 2.2 were performed over a dedicated local area network with no loss. Tests were performed between two machines, A and B, as shown in Figure 2.8. The round trip time was on average 10ms, and no background traffic was present.

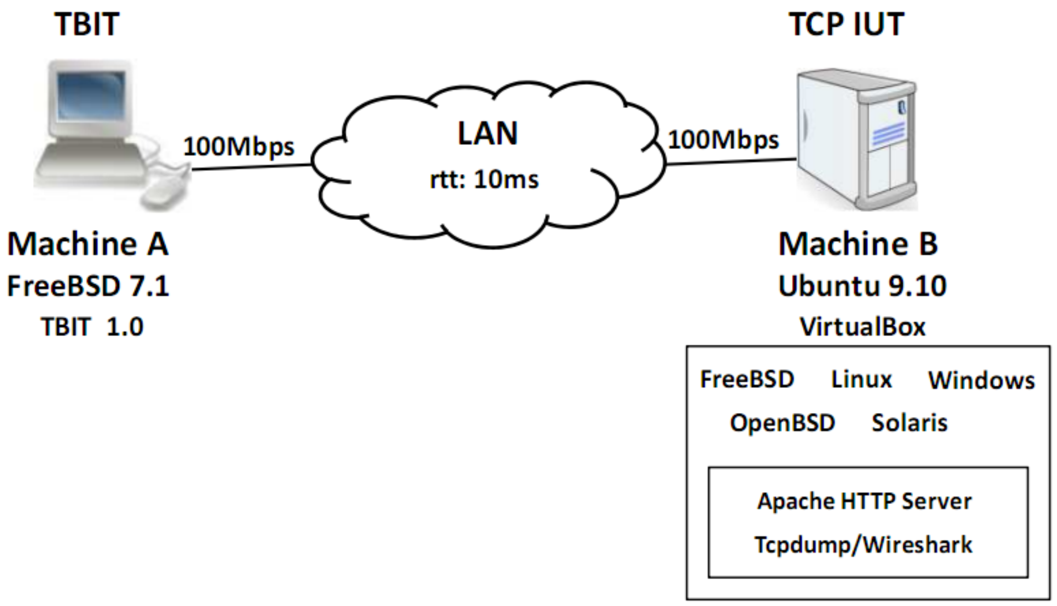


Figure 2.8: Experimental design for TBIT testing

The IUTs being verified were the standard TCP stacks of various operating systems. We installed 27 operating systems using Oracle's VirtualBox virtualization software [Virtualbox] on machine B. We ran tests for Mac OS X on another machine.

TBIT 1.0 [Tbit] was extended on FreeBSD 7.1 (machine A) with the seven TBIT tests detailed in the Section 2.2.

For each operating system, we installed an Apache HTTP Server [Apache] on machine B since TBIT is originally designed to infer TCP behavior of a web server. The TCP segments transmitted between TBIT and each IUT were captured at machine B. For this purpose, we also installed wireshark [Wireshark] on each Windows OS, and tcpdump [Tcpdump] on each UNIX, UNIX-like and Mac OS.

2.4 Results of TCP Behavior Inference Tool (TBIT) Testing

We verified the operating systems in Table 2.1. Each TBIT test was repeated three times. In every case, all seven test outputs were consistent. Segment captures of tests and TBIT tests are available [Ekiz 2011c].

For test A, the early versions of FreeBSD, 5.3 and 5.4, and all versions of OpenBSD report at most three SACK blocks (Misbehavior A1). OpenBSD explicitly defines a parameter TCP_MAX_SACK = 3. Windows 2000, XP and Server 2003 report at most two SACK blocks (Misbehavior A2). Later Windows versions correct this misbehavior.

If the return path carrying SACKs were lossless, a TCP data receiver reporting at most two or three SACK blocks would not cause a problem. A data sender would always infer the proper state of the receive buffer for efficient SACK-based loss recovery described in [RFC3517]. When more than four SACK blocks exist at a data receiver, and SACK segments are lost, the chance of a data sender getting less accurate state of the receive buffer increases as SACK implementations' number of blocks reported is decreased. This misbehavior can lead to less efficient SACK-based loss recovery, and therefore decreased throughput (longer transfer times) when multiple TCP segments are lost within the same window.

Table 2.1: TBIT test results

Operating System	Test							
	A1	A2	B	C	D	E	F	G
FreeBSD 5.3	X				X			
FreeBSD 5.4	X				X			
FreeBSD 6.0								
FreeBSD 7.3								
FreeBSD 8. 0								
Linux 2.2.20 (Debian 3)							X	
Linux 2.4.18 (Red Hat 8)							X	
Linux 2.4.22 (Fedora 1)							X	
Linux 2.6.12 (Ubuntu 5.10)							X	
Linux 2.6.15 (Ubuntu 6.06)							X	
Linux 2.6.18 (Debian 4)							X	
Linux 2.6.31 (Ubuntu 9.10)								
Mac OS X 10.5								
Mac OS X 10.6								
OpenBSD 4.2	X				X			
OpenBSD 4.5	X				X			
OpenBSD 4.6	X				X			
OpenBSD 4.7	X				X			
OpenBSD 4.8	X				X			
OpenSolaris 2008.05							X	X
OpenSolaris 2009.06							X	X
Solaris 10								X
Solaris 11							X	
Windows 2000		X	X	X	X	X		
Windows XP		X	X	X	X	X		
Windows Server 2003		X	X	X	X	X		
Windows Vista					X	X		
Windows Server 2008					X	X		
Windows 7					X	X		

We report, for test B, that Windows 2000, XP and Server 2003, are misbehaving. SACK information is not reported where it should be, after the cumulative ACK is increased beyond the first SACK block. Later Windows versions correct this misbehavior.

Misbehavior C is observed with Windows 2000, XP and Server 2003. SACK information is partially reported when the data between two previously reported SACK blocks are received. Later Windows versions correct this misbehavior.

We observed misbehavior D, failure to report SACK information in FIN segment, in FreeBSD 5.3, FreeBSD 5.4, all versions of OpenBSD and Microsoft's Windows. The problem has been corrected in the later FreeBSD versions.

Misbehavior E is observed with all versions of Windows OS. When the TCP traffic is bidirectional, SACKs are not carried within the opposite direction TCP segments. Out-of-order data are SACKed only once when they arrive. If a SACK is lost on the return path, subsequent segments with no SACKs will trigger a fast retransmission which can cause the data sender to unnecessarily retransmit data that as SACKed and already exists in the receiver's buffer.

The traffic pattern for testing misbehavior E is a typical web browsing scenario. TBIT represents a user's web browser where HTTP 1.1 GET requests are pipelined, and the IUT represents an HTTP 1.1 web server. Since the scenario represents typical Internet traffic, we believe that the SACK generation misbehavior of the Windows OS is significant, and should be fixed.

Misbehavior F is observed in Solaris 11, OpenSolaris and all Linux systems except the latest one tested Linux 2.6.31 (Ubuntu 9.10), so the problem may be fixed for Linux. Interestingly, misbehavior F did not occur in Solaris 10. When out-of-order data exists at the data sender, thus sending both data payload and SACKs, every other segment carries only bytes equal to SACK information appended (at most 36 bytes). This misbehavior halves the throughput for the time out-of-order data exists at the receive buffer, and is the typical web browsing scenario described above. We consider the misbehavior significant, and needs to be fixed.

Misbehavior G is observed on Solaris 10 and OpenSolaris systems. We ran the Test G multiple times with different time intervals X = {1, 5, 15} minutes. Even after 15 minutes, we frequently observed the reappearance of SACK blocks from a prior connection in later connections. The SACK-based loss recovery algorithm does not work efficiently, when the TCP implementation has this misbehavior. For example, when two connections have overlapping sequence numbers, the latter connection sends a SACK for a data block that was never received. This misbehavior will cause a decrease in throughput. We would like to note that the ISN of a new TCP connection is assigned randomly and the probability of having two TCP connections using the same ISN space is small.

One time, we ran all the seven TBIT tests continuously on Solaris 10 and OpenSolaris machines, and noticed a scenario where a SACK block of the first connection in Test A appeared in the SYN-ACK segment of the *third* connection established in Test C. One time, all TBIT tests were executed and then repeated 45 minutes later. Even after 45 minutes, we observed an instance where the SACK block of Test E from the first set appeared in the SYN-ACK segment of Test E in the second set. We could not repeat this misbehavior with any regularity. Having a sender think data is acknowledged when in fact the data has not been received results in an inconsistent (i.e., unreliable) state. Fortunately, this misbehavior is corrected in Solaris 11.

2.5 Conclusion

In this research, we designed a methodology and verified conformant SACK generation on 29 TCP stacks for a wide range of OSes: FreeBSD, Linux, Mac OS X, OpenBSD, Solaris and Windows. We identified the characteristics of the seven misbehaviors, and designed seven new TBIT tests to uncover these misbehaviors.

For the first five misbehaviors which are observed in the CAIDA trace files, we found at least one misbehaving TCP stack. We report various versions of OpenBSD and Windows OS to have misbehaving SACK generation implementations. In general, the misbehaving SACK implementations can cause a less efficient SACK-based loss recovery which yields to decreased throughput and longer transfer times.

During the TBIT testing, we identified two additional misbehaviors (F and G). Misbehavior F decreases the throughput by sending less than expected data while using SACKs. Most Linux and OpenSolaris systems show this misbehavior. Misbehavior G is more serious and can cause a TCP connection to be inconsistent should the sequence number space of one connection overlap that of a prior connection. Solaris 10 and OpenSolaris systems misbehave in this manner.

We note that for all misbehaviors, because SACKs are advisory thus allowing a data receiver to renege on all SACKed out-of-order data, eventually the data sender-receiver will timeout, discard all SACK information, and return to a correct state. Thus the data flow remains reliable; only performance degradation may occur.

As stated in the Introduction (Section 2.1), we discovered SACK misbehaviors during our investigation of data reneging described in Chapter 3. In that investigation, we argue that SACKs should be “permanent” (not advisory) meaning a data receiver MUST NOT renege on out-of-order data. If SACKs were to become permanent, misbehavior G would have to be fixed since it can result in unreliable data transfer. While we hope misbehaviors A-F will be fixed, even if left as is, as long as SACKs remains advisory the misbehaviors will only result in reduced performance, not unreliable data transfer.

While simple in concept, SACK handling is complex to implement.

Chapter 3

DETECTING RENEGING THROUGH PASSIVE MEASUREMENT

To document the frequency of TCP reneging, a mechanism is needed to detect reneging instances. This section presents a model and its implementation as a tool, RenegDetect, which passively detects TCP reneging instances occurring in Internet traces. TCP does not support detecting reneging at a data sender. On the other hand, SCTP supports detecting reneging at a data sender. When previously SACKed data are not SACKed in a new acknowledgement (ack), an SCTP data sender infers reneging. Our model to detect TCP reneging instances 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. When an inconsistency occurs between the state of the receive buffer and a new ack, reneging is detected. We implemented the model as a tool called RenegDetect and tested RenegDetect with artificial TCP flows mimicking reneging instances.

RenegDetect was also verified by analyzing 100s of TCP flows from Internet traces. The analysis showed that reneging was happening frequently. On closer inspection, however, it turned out that reneging was not happening, rather the generation of SACKs in many TCP implementations was incorrect according to [RFC2018]. Some TCP implementations were generating SACKs incompletely under some circumstances. Sometimes the SACK information that should have been sent was not. Sometimes wrong SACK information was sent. We refer to these implementations as misbehaving. In Internet traces, we observed five different types of misbehaviors which wrongly gave the impression that reneging was occurring. Our discovery led us to a side investigation to precisely identify the five misbehaving TCP stacks observed in the CAIDA traces. We tested [RFC 2018] conformant SACK generation on a wide range of operating systems and found at least one misbehaving TCP stack (see Section 2.4 for more detail) for each of the five misbehaviors.

Discovering TCP SACK generation misbehaviors led us to change our initial method to detect reneging instances which was based only on monitored acks. In addition to acks, RenegDetect was extended to analyze the flow of data, in particular, retransmissions of data which are a more definitive indication that reneging has occurred.

Initially, our hypothesis was that reneging rarely if ever occurs in practice. To statistically conclude with confidence that reneging is a rare event, say P(reneging) < 10^{-5}, we needed to analyze ~300K TCP connections using SACKs and document that no instances of reneging occurred. For that purpose, TCP traces from three different domains (Internet backbone, wireless, enterprise) were analyzed using our updated RenegDetect to report the frequency of reneging.

Contrary to our initial hypothesis that reneging rarely if ever occurs in practice, trace analysis demonstrated that reneging does happen. For the reneging instances detected, we predicted reneged hosts operating systems with TCP fingerprints and characterized reneging behavior in detail.

The outline of this chapter is as follows. First, we detail how a TCP or SCTP data sender infers reneging in Section 3.1. In Section 3.2.1, we present our initial model (RenegDetect v1) to detect reneging instances in the Internet traces. Section 3.2.2 describes the validation of our original RenegDetect v1 using artificial TCP flows and real TCP flows from the Internet. Internet trace analysis demonstrated that reneging is inferred wrongly due to SACK generation misbehaviors. Modifications made to RenegDetect to create v2 to infer reneging and misbehavior instances more accurately via data retransmissions are explained in Section 3.2.3. Section 3.2.4 details the probability theory to define a minimum number of TCP flows to analyze to test our initial hypothesis. We report the frequency of reneging in Section 3.2.5. Finally, Section 3.3 concludes our efforts.

3.1 Detecting Reneging at TCP and SCTP Data Senders

This section details reneging behavior of a data sender in reliable transport protocols. Reneging is possible both in TCP and SCTP. To generalize reneging behavior in the Internet, the frequency of reneging for both protocols should be documented. Unfortunately, SCTP is not sufficiently deployed to matter. TCP is the dominant protocol used for reliable data transfers in the Internet. Thus, to generalize reneging behavior, we need to document frequency of TCP reneging. For that, a mechanism is needed to detect reneging instances in the Internet traces. If a TCP sender had a mechanism to detect reneging, we could simply replicate that mechanism and apply the mechanism to TCP traces. Unfortunately, a TCP sender does not support detecting reneging. Instead, a TCP sender tolerates reneging with a retransmission policy specified in [RFC2018]. An SCTP data sender, on the other hand, supports detecting reneging by keeping a state for previously SACKed data. This state is compared to SACK information carried within new acks and reneging is detected when a comparison is inconsistent. We borrow SCTP's approach and apply it to detect TCP reneging instances within Internet traces. For that, we detail how an SCTP data sender detects reneging with an example. We first present how a TCP data sender tolerates reneging and possible limitations which might be reasons for not detecting reneging at a TCP data sender.

In the current TCP and SACK specifications, a TCP data sender has no design to infer reneging. To tolerate reneging, a TCP data sender keeps copies of SACKed data in its send buffer until cumulatively ACKed. To achieve reliable data transfer, the following retransmission policy is specified in [RFC2018] for a data sender to resume the data transfer in the case of reneging.

For each segment in the send buffer that is SACKed, an associated “SACKed” flag is set. The segments with “SACKed” bit set are not retransmitted until a retransmission timeout (RTO). At the RTO, the TCP data sender clears all the “SACKed” information due to possible reneging and begins retransmitting all segments beginning at the left edge of the send buffer.

A TCP data sender’s lack of inferring reneging (a retransmission policy is specified to tolerate reneging instead) might be due to the following SACK limitations. First, there is a hard limit on the number of SACK blocks that can be acknowledged based on the constrained space in the TCP options field. At most, 4 SACK blocks can be reported in a TCP segment if no other TCP options are used. Second, a data sender may not infer if a segment is “SACKed” or not when four consecutive SACKs are lost on the ack path. These limitations prohibit a TCP data sender from having an accurate view of the data receiver’s receive buffer state to detect reneging.

SCTP, on the other hand, supports reneging detection at the data sender. Unlike TCP’s constrained number on reported SACK blocks (4 at maximum), an SCTP data receiver can generate SACK chunks with a large number of Gap Ack

Blocks (same semantics as SACK blocks). In SCTP [RFC4960], a data receiver must report as many Gap Ack Blocks as possible in a SACK chunk. While a limit still exists on the number of reported blocks restricted by the path's maximum transmission unit (MTU) for practical purposes, the limit does not come into play. For example, for a path with MTU=512 bytes, a SACK chunk can report 116 Gap Ack Blocks (20 bytes for an IP header, 12 bytes for a SCTP common header, 16 bytes for a SACK chunk header + 116 * 4 byte Gap Ack Blocks).

Thus, an SCTP data sender has a more accurate view (effectively complete) of the data receiver's buffer state, and can accurately infer reneging by inspecting reported Gap Ack Blocks[1]. If a new SACK arrives and previously SACKed data is not present, the SCTP data sender infers reneging, and marks only the reneged data for retransmission.

Let us look at an example reneging scenario shown in Figure 3.1 and see how an SCTP data sender infers reneging in detail. Without loss of generality, the example assumes 1 byte of data is transmitted in each data packet. A data sender sends a sequence of packets, 1 through 6, to a data receiver. Assume packet 2 is lost in the network. The data receiver receives packets 3 through 6, and sends ACKs and SACKs to notify the data sender about the out-of-order data received. When ACK 1 SACK 3-6 arrives at the data sender, the state of the receive buffer is known to be as follows: ordered data 1 is delivered or deliverable to the receiving application, and out-of-order data 3-6 are in the receive buffer.

[1] The likelihood of a data stream requiring more than 116 out-of-order blocks of data is negligible.

Before packet 2 is retransmitted via a fast retransmission, assume the operating system running the data receiver runs short of main memory, and reneges all out-of-order data in the receive buffer. When packet 2's retransmission arrives at the data receiver, only a cumulative ACK 2 is sent back to the data sender with no SACKs.

When the data sender receives ACK 2, reneging is detected. Previously SACKed out-of-order data 3-6 are still not being SACKed. Data 3-6 are marked for retransmission as the data sender infers reneging.

ACK 2 SACK 7-7 is sent when packet 7 arrives out of order. This SACK reinforces the fact that reneging (for data 3-6) occurred.

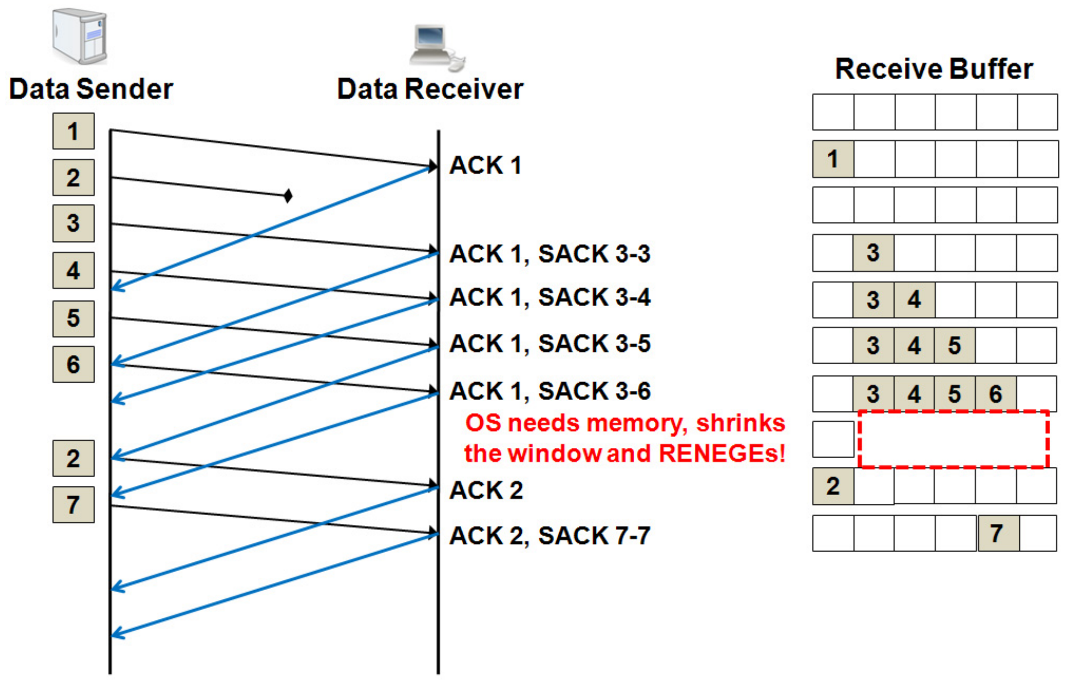


Figure 3.1: Detecting reneging at the SCTP data sender

To report the frequency of TCP reneging, a mechanism to detect TCP reneging instances is needed. We next present a method to detect reneging instances which is based on how an SCTP data sender infers reneging.

3.2 Detecting Reneging in the Middle

To document the frequency of TCP reneging, a mechanism is needed to detect reneging instances. This section presents a model and its implementation as a tool, RenegDetect, which passively detects TCP reneging instances occurring in Internet traces. In passive measurement studies, collected trace files are analyzed to infer a specific protocol behavior (i.e., reneging). Our model infers reneging instances by analyzing TCP acknowledgment traffic monitored at an intermediate router. The model is based on how an SCTP data sender infers reneging. When previously SACKed data are not SACKed in a new ack, reneging is inferred. The model is detailed in Section 3.2.1. We implemented the model as a tool called RenegDetect v1.

RenegDetect v1 was verified with artificial TCP flows mimicking reneging instances. The tool to validate the correctness of RenegDetect v1 is presented in Section 3.2.2. RenegDetect v1 was also verified with 100s of TCP flows from Internet traces. Preliminary Internet trace analysis showed that reneging albeit infrequent was happening. Upon deeper investigation, we revealed that some TCP stacks were generating partial or wrong SACKs falsely giving the impression that reneging was happening. Discovering that misbehaving SACK implementations exist led us to update our model.

Our initial model infers reneging instances by analyzing acks. To detect reneging instances more accurately, our original RenegDetect v1 was updated to analyze flow of data too, specifically data retransmissions. Section 3.2.3 details changes to RenegDetect v2.

Once RenegDetect v2 was ready to analyze TCP traces, we needed to find the minimum number of TCP flows for analyzing to statistically conclude that reneging is a rare event, say P(reneging) < 10^{-5}. Section 3.2.4 details the probability theory to determine the number of TCP flows to be analyzed for this investigation. To confirm our initial hypothesis that reneging is a rare event, we needed to analyze ~300K TCP connections using SACKs.

Finally, TCP traces from three different domains (Internet backbone, wireless, enterprise) were analyzed using the RenegDetect v2 to report the frequency of reneging. The results of trace analysis are presented in Section 3.2.5.

3.2.1 The Model (RenegDetect v1)

This section details a model and its implementation, RenegDetect v1, which detects TCP reneging instances using the TCP trace files. The model described in this section appears in [Ekiz 2010]. The model 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 through new acks. When an inconsistency occurs between the state of the receive buffer and a new ack, reneging is detected.

A data receiver's receive buffer consists of two types of data: ordered data, which has been ACKed but not yet delivered to the application, and out-of-order data that resulted from loss or reordering in the network. To detect an SCTP reneging instance, a data sender infers the state of the receiver's buffer through ACKs and SACKs. Even though TCP does not have a mechanism to detect reneging instances, reneging can be detected by analyzing TCP ack traffic and inferring the state of receiver's buffer. The idea is to learn the state of the receive buffer and to employ a similar reneging detection mechanism as an SCTP data sender does based on the observed acks. From now on, all discussions regarding to detecting reneging instances apply only to TCP traffic.

For a TCP data sender, the state of the receive buffer can be learned with the ACKs and SACKs, and updated through the new acks. The state consists of two items: a cumulative ACK value (stateACK) and a list of out-of-order data blocks (stateSACKs) known to be in the receive buffer.

Now let us briefly describe how the state of the receive buffer is maintained and reneging is inferred at the data sender with the help of the reneging example shown in Figure 3.1. Assume that all acks sent by a TCP data receiver arrive at the corresponding TCP data sender.

The first ack, ACK 1, indicates that no out-of-order data are in the receive buffer. The state of the receive buffer is initialized as follows: ordered data 1 is delivered or deliverable to the receiving application (stateACK is set to 1) and no out-of-order data are in the receive buffer (no stateSACK blocks). The next ack, ACK 1 SACK 3-3, notifies that out-of-order data 3 is received and stored in the receive buffer. This ack updates the state of the receive buffer: ordered data 1 is delivered or deliverable to the receiving application (stateACK is still 1) and out-of-order data 3 is in the receive buffer (add the first stateSACK 3-3 to the state).

When the acks for packets 4-6 are each observed, the state of the receive buffer is updated and the out-of-order data 3-6 are known to be in the receive buffer. The state of the receive buffer is now: stateACK 1, stateSACK 3-6. The next ack, ACK 2, arrives with no SACK blocks (assuming there is enough space in the TCP segment to report at least one SACK block). When the state of the receive buffer (stateACK 1, stateSACK 3-6) is compared to the new ack (ACK 2), an inconsistency is observed. The cumulative ACK informs that data up to 2 are delivered or deliverable to the receiving application and no out-of-order data are in the receive buffer. On the other hand, it is known that out-of-order data 3-6 have been previously SACKed (stateSACK 3-6). So, due to the lack of a SACK block for the out-of-order data 3-6, reneging is detected.

Let us consider the example scenario when the ack traffic is monitored by an intermediate router. In the example shown in Figure 3.1, a reneging instance is detected when all of the acks arrive at the data sender. In practice, acks may traverse different paths, arrive at the intermediate router out of order, or get lost in the network before reaching the router.

Figure 3.2 shows the same data transfer where only three acks are monitored at the intermediate router. Upon seeing ACK 1 SACK 3-4, the router determines that the state of receive buffer at the data receiver is: 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). The second ack, ACK 1 SACK 3-6, updates the 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 the receive buffer (stateACK 1, stateSACK 3-6), an inconsistency is observed. Reneging is detected since previously SACKed data 3-6 are not SACKed.

Even though the number of acks observed at the intermediate router are limited, the state of the receive buffer is as for Figure 3.1. Because a SACK block reports all of the consecutive out-of-order data as a single block, the intermediate router can infer the complete state of the receive buffer most of the time.

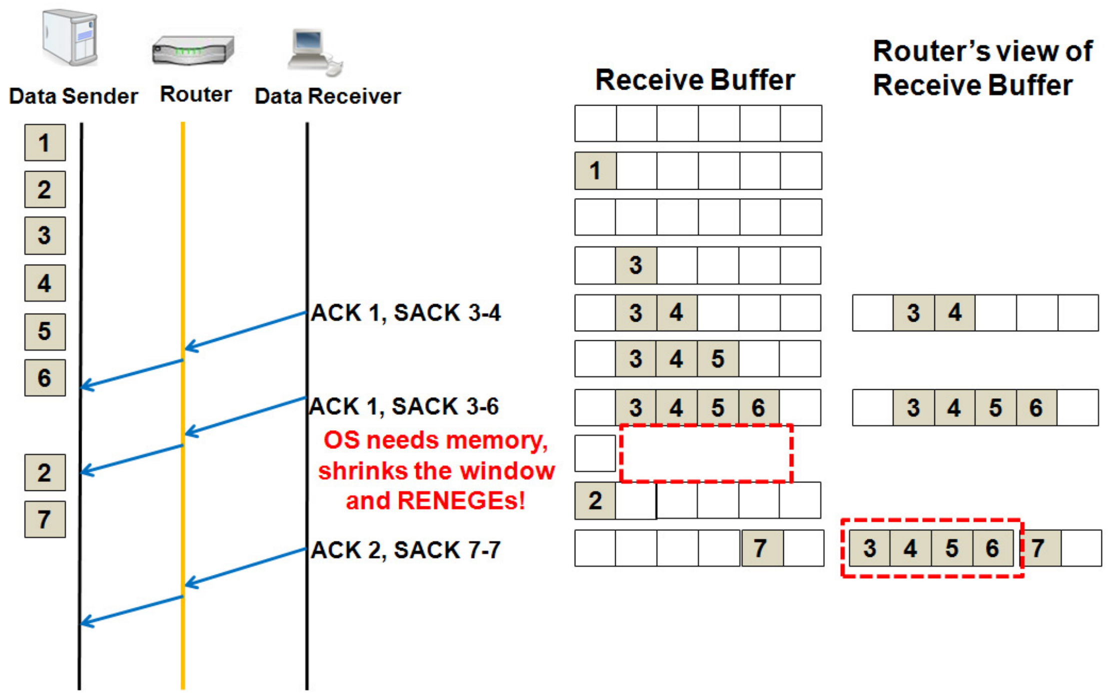


Figure 3.2: Detecting reneging at an intermediate router

Constructing the state of the receive buffer as accurately as possible is based on the actual number of SACK blocks at the data receiver. If the number of SACK blocks is more than 4, the data receiver is unable to report full SACK information. In this case, when consecutive acks get lost, the intermediate router will have less accurate state information.

Table 3.1 shows the number of SACK blocks in TCP segments based on a few randomly selected trace files from the Internet backbone captured in June 2008. Recall that, at maximum, 4 SACK blocks can be included in a TCP segment. For segments with 1, 2, or 3 SACK block(s), the TCP header length is checked to determine if another SACK option could have been appended to a TCP header. TCP segments with 4 SACK blocks already have a full TCP header. Less than 0.5% of the TCP segments that include SACK options do not have enough space for another SACK option. Assuming all TCP traces follow a similar pattern, the state of the receive buffer can be constructed accurately most of the time.

Even though the state of receive buffer may be inaccurate, having a partial state of the out-of-order data in the receive buffer can be still enough to detect reneging instances. The reasoning is that we expect that a reneging data receiver will purge all of the out-of-order data, as it occurs in FreeBSD [FreebsdImpl] and Linux [Linux]. Since the intermediate router has state information about out-of-order data, reneging will be detected as soon as any ack with no SACK option is observed.

Table 3.1: Number of SACK blocks in TCP segments

TCP segments with *n* SACK blocks	Enough space for another SACK block	Not enough space for another SACK block
1	~88%	0%
2	~11%	0%
3	0.7%	0.20%
4	n/a	0.15%
Total number of TCP segments	780,798 (100%)	

Our software to detect reneging instances, RenegDetect v1, constructs the state of the receive buffer for TCP flows (connections) that contain SACKs. An inferred state of the receive buffer is compared with new acks to check for consistency. When the comparison is consistent, the state is updated. Otherwise, a data reneging instance is detected and reported.

We now describe our model for constructing the state of the receive buffer at an intermediate router. The state consists of a cumulative ACK (stateACK) value and an ordered list of out-of-order data blocks (stateSACK blocks) known to be in the receive buffer. In Figure 3.3, a view of the receive buffer state is shown, which consists of *n* disjoint stateSACK blocks. The stateSACKs are ordered according to the sequence number of their left edges.

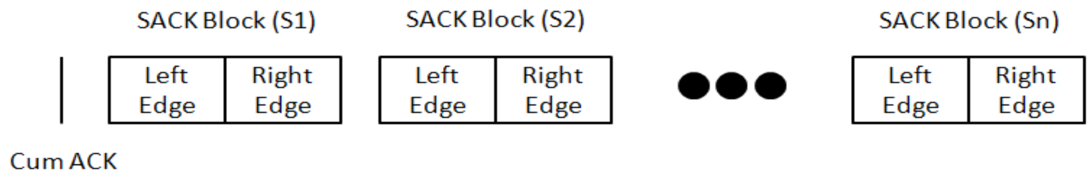


Figure 3.3: Receive buffer state

The stateACK value holds the highest ACK value observed for the TCP flow and is updated when a higher ACK value is observed. When the stateACK value is updated, any stateSACKs below the stateACK value are deleted from the state.

RenegDetect v1 currently does not deal with wrap around in the sequence space simply to avoid programming complexity for such a rare event. If the stateACK value is higher than any of the stateSACKs due to a wrap around a warning is thrown and the given TCP flow is simply discarded.

Figure 3.4 presents our model for constructing and updating the stateSACKs in the state of the receive buffer. The state is initialized with the first TCP ack observed in a flow. If the ack has no SACK block(s), only the stateACK is initialized. If the ack includes SACK blocks(s), each one is added as a stateSACK to the state.

When the next TCP ack is observed, each reported SACK block (corresponding to a New SACK Block (N) in Figure 3.4) is compared with the stateSACKs in the receive buffer state. Each stateSACK block in the receive buffer state is represented with Current SACK Block (C) in Figure 3.4.

The comparison of a new SACK block (N) and a current SACK block (C) is done both on the left (L) and right (R) edges. If each SACK block is thought of as a set, a comparison of two sets must result in exactly one of four cases:

1. N is a superset of C $(N \supseteq C)$
2. N is a proper subset of C $(N \subset C)$
3. N intersects with C, and N and C each have at least 1 byte of data not in C and N, respectively $((N \cap C \neq \emptyset) \land !(N \supseteq C) \land !(N \supset C))$
4. N does not intersect with C $(N \cap C = \emptyset)$

Note that the above cases are all mutually exclusive. Each case is described in turn. For the given examples, assume an initial receive buffer state as follows: the stateACK is 8 and there is one stateSACK 12-15 with left and right edges 12 and 15, respectively.

Case 1: When a new SACK block (e.g., SACK 12-17) is a superset of a current SACK block (e.g., stateSACK 12-15), it means more out-of-order data had been received at the data receiver. The current SACK block (stateSACK) is updated to reflect the new SACK block (information). The update may be in terms of a left edge extension, a right edge extension, or both. After the update, the updated stateSACK is compared with the rest of the stateSACKs in the state. The reasoning is that the updated stateSACK may be the superset of a number of stateSACKs in the receive

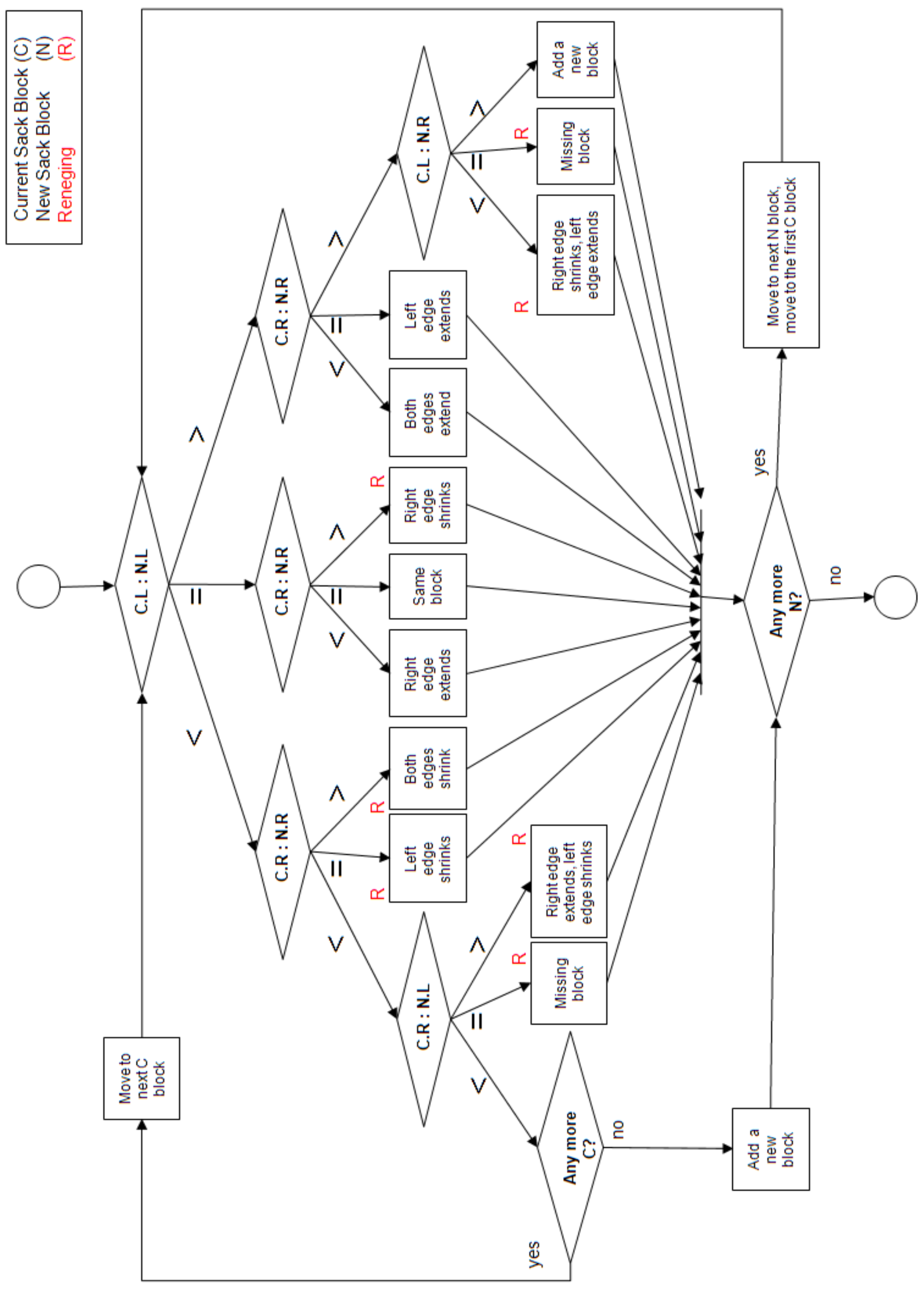


Figure 3.4: Reneging detection model

buffer state due to possible ack reordering, and may fill a gap between two stateSACKs. Now assume that a new ack, ACK 8 SACK 12-17, arrives. When C and N are compared, case 1 holds. C is updated via a right edge extension to reflect the new information learnt from N; stateSACK becomes 12-17.

Case 2: When a new SACK block (e.g., SACK 12-13) is a proper subset of a current SACK block (e.g., stateSACK 12-17), the comparison implies reneging (out-of-order data 14-17 had been deleted from the receive buffer). An instance of reneging is logged for future deeper analysis.

Case 3: Reneging is similarly detected when a new SACK block (e.g., SACK 15-20) intersects with a current SACK block (stateSACK 12-17), and the new SACK block and current SACK block each have at least 1 byte not in the current SACK block and new SACK block, respectively. Such a case would result when a data receiver purges some, but not all, of the out-of-order data and later receives more out-of-order data. The new ack informs the arrival of new out-of-order data, 18-20, as well as the removal of previously SACKed data, 12-14. An instance of reneging is logged for future deeper analysis. The state is not updated (to catch more inconsistencies) until the cumulative ACK is advanced beyond the SACK blocks that trigger reneging instances.

Case 4: If a new SACK block (e.g., SACK 22-25) and a current SACK block (e.g., stateSACK 12-17) do not intersect, the new SACK block is compared with the next stateSACK block in the state. If the new SACK block is disjoint with all of the stateSACKs in the state, the new SACK block is added as a stateSACK to the receive buffer state. The updated receive buffer state becomes: stateACK 8, stateSACK$_1$ 12-17, stateSACK$_2$ 22-25. If a new ack reports only one SACK block, say ACK 8 SACK

22-25, and there is no space in the TCP header to append another SACK block, case 4 holds. If the new ack has enough space to carry two SACKs (22-25, 12-17) but carries only one (22-25), RenegDetect detects an inconsistency in the state: previously SACKed data 12-17 is missing. An instance of reneging for out-of-order data 12-17 is logged for future deeper analysis.

The model, shown in Figure 3.4, detects reneging instances only when some SACK blocks are included in the acks. If the data receiver purges all the out-of-order data in the receive buffer, no SACK blocks are reported within acks. In such a case, the receive buffer state would have a number of stateSACKs, and the new ack would report no SACK blocks (even though the TCP options field has enough space to report at least one SACK block). RenegDetect v1 also infers such reneging instances. Let us continue with the example scenario. The receive buffer state is as follows: stateACK 8, stateSACK_1 12-17, and stateSACK_2 22-25. A new TCP ack arrives with no SACK blocks (ACK 8). Reneging is detected if there is enough space in TCP header to report at least one SACK block.

Reneging may be inferred spuriously if acks are reordered before arriving at the intermediate router. To cope with ack reordering, a check is performed on the protocol fields: IP ID and TCP ACK. When one or both of the fields of an ack is smaller than the previous ack's values, reordering is detected. Reordered acks are not used to update the receive buffer state; they are discarded.

We also considered looking at TCP timestamps [RFC1323] to cope with ack reordering. Unfortunately, Internet TCP traces show that acks from the same window may have the same TCP timestamp value. On the other hand, IP ID field is always incremental. As such, we chose to use IP ID field along with the TCP ACK field to identify reordering.

We needed to test if RenegDetect v1 can detect reneging instances correctly before analyzing real TCP traces. A tool to validate RenegDetect v1 is explained in the following section.

3.2.2 Validating RenegDetect v1

A validation tool was needed to check whether or not RenegDetect v1 could identify reneging instances correctly. For that purpose, another student from our lab, Abuthahir Habeeb Rahman, independently created a number of synthetic TCP flows carrying ack traffic to simulate some reneging and non-reneging flows. He used text2pcap, an application that comes with the Wireshark protocol analyzer [Wireshark] which can generate a capture file from an ASCII hex dump of packets.

Reneging flows mimicked behaviors such as: (1) a SACK block was shrinking from left edge, right edge, or both, (2) only one SACK was reported when two SACK blocks were expected, (3) a SACK block was shrinking from one edge while extending from the other, and (4) an ACK was increasing into the middle of a SACK block.

Non-reneging flows mimicked behaviors such as: (1) a SACK block was extending from left edge, right edge, or both, (2) a new SACK block was covering previous two SACKs, and (3) an ACK was increasing to the right edge of a SACK block or beyond.

RenegDetect v1 was tested with these synthetic flows. All of the reneging instances were correctly identified.

RenegDetect v1 was also verified by analyzing 100s of TCP flows from Internet traces provided by Cooperative Association of Internet Data Analysis (CAIDA). Initially, it seemed that reneging was happening frequently. On closer inspection, however, it turned out that the generation of SACKs in many TCP implementations was incorrect according to [RFC2018]. Some TCP implementations were generating SACKs incompletely under some circumstances. Sometimes the SACK information that should have been sent was not. Sometimes wrong SACK information was sent. These misbehaviors wrongly gave RenegDetect v1 the impression that reneging was occurring.

Our discovery led us to a side investigation to confirm whether or not the misbehaviors observed in the CAIDA traces were actual reneging instances or misbehaving TCP stacks. We tested [RFC 2018] conformant SACK generation on a wide range of operating systems. In our testing, we simply mimicked the traffic behavior observed in the CAIDA traces prior to observed misbehaviors. For the five misbehaviors observed in the CAIDA traces, we found at least one misbehaving TCP stack (see Section 2.4 for more detail). This discovery led us to change the way RenegDetect v1 detected reneging instances. We explain our updated model and tool in the next section.

3.2.3 RenegDetect v2 (with Misbehaviors Detection and Use of Bidirectional Traffic)

Based on the verification described in Section 3.2.2, we needed to update our model to detect reneging. The problem was how to differentiate between an actual reneging instance vs. a SACK generation misbehavior.

We decided to distinguish misbehavior and reneging instances based on the monitored data retransmissions. In misbehaviors, out-of-order data are not discarded from the receive buffer. Only related SACK information is missing or reported partially. Eventually, when the data between ACK and the out-of-order data are received, ACK is increased beyond the out-of-order data that seemed to have been reneged. If no or partial retransmissions are monitored for the out-of-order data that seemed to have been reneged and ACK is increased beyond, we conclude that a misbehavior is observed (no reneging).

On the other hand, out-of-order data are discarded with reneging. Therefore, 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, 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. Let us show how to detect reneging by analyzing retransmissions with an example shown in Figure 3.5. The example is similar to example shown in Figure 3.1 except that transmission sequence of packets 7 and 2 is exchanged and data retransmissions for packets 3-6 are present. Before packet 7 is received, the data receiver reneges and deletes out-of-order data 3-6. When packet 7 is received, an ack (ACK 1 SACK 7-7) is sent. When ACK 1 SACK 7-7 is compared to the state (stateACK 1 stateSACK 3-6), an inconsistency exists. Previously SACKed data 3-6 are not SACKed anymore 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 was increased to 7 on receipt of packet 2, this behavior would conclude a SACK generation misbehavior (no retransmissions). After an RTO, the data sender retransmits packets 3-6. Since ACK increases steadily after each retransmission, reneging is concluded.

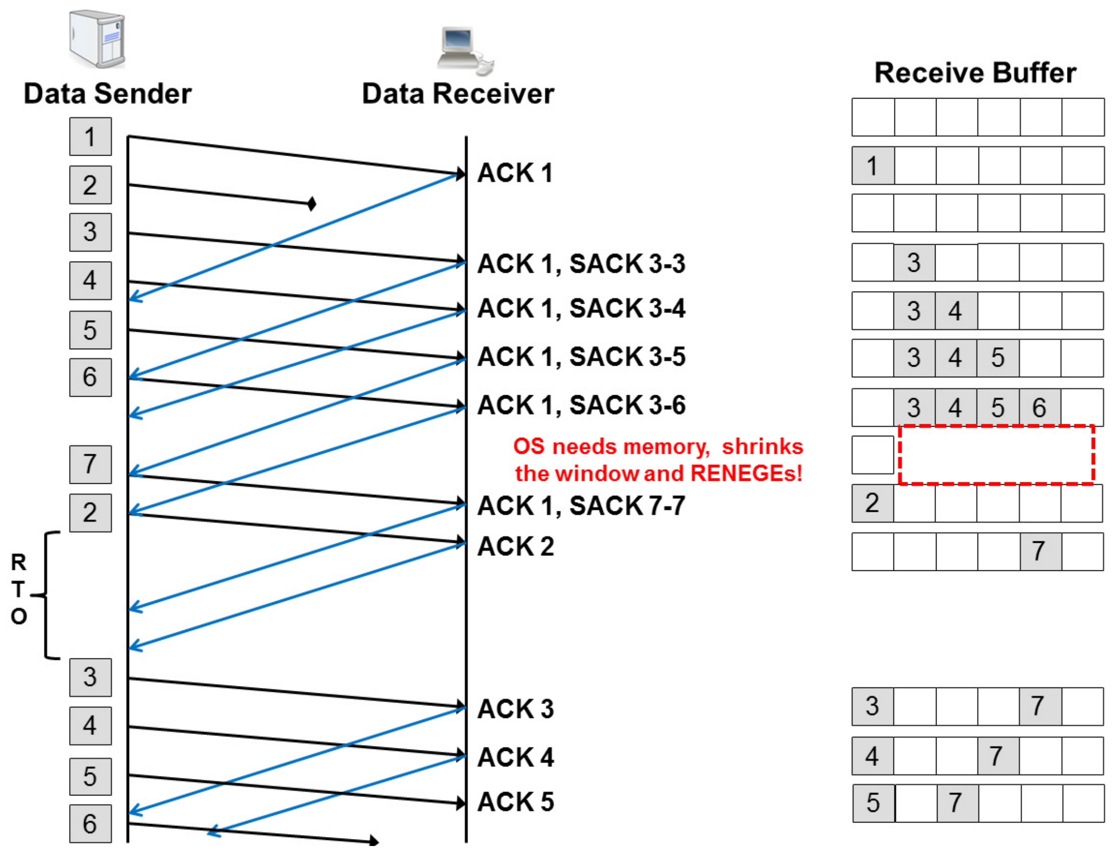
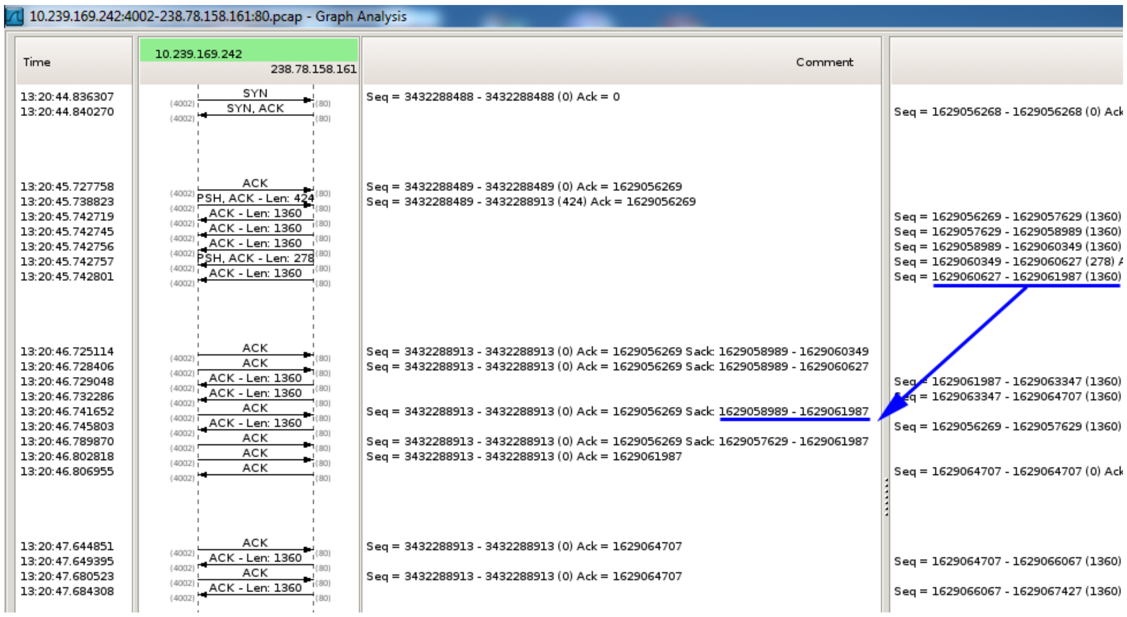


Figure 3.5: Detecting reneging by analyzing retransmissions

Does observing retransmissions for data that seems to have been reneged assure a reneging instance? No. When retransmissions are observed for the out-of-order data that seems to have been reneged (referred to as “a candidate reneging instance”) three cases are possible: (I) a not reneging instance (a misbehavior), (II) an ambiguous instance (either a reneging or a misbehavior instance), or (III) a reneging instance. RenegDetect v2 reports candidate reneging instances. We then analyzed each candidate reneging instance by hand with wireshark [Wireshark]. Wireshark can graph a TCP flow displaying both data and ack segments. Initially, wireshark did not have support the viewing of SACK blocks. A student from our lab, Fan Yang, extended wireshark to display SACK blocks in a flow graph. The patch to view SACKs in wireshark flow graphs can be downloaded at: http://www.cis.udel.edu/~amer/PEL/Wireshark_TCP_flowgraph_patch.tar. An example output is shown in Figure 3.6 where the underlined data are shown in a SACK block indicated by an arrow. With this update to wireshark, it easy to analyze a TCP flow and decide which case holds for a candidate reneging instance.



```
10.239.169.242:4002-238.78.158.161:80.pcap - Graph Analysis

Time             10.239.169.242 / 238.78.158.161   Comment
13:20:44.836307  SYN ->                            Seq = 3432288488 - 3432288488 (0) Ack = 0
13:20:44.840270  <- SYN, ACK                       Seq = 1629056268 - 1629056268 (0) Ack

13:20:45.727758  ACK ->                            Seq = 3432288489 - 3432288489 (0) Ack = 1629056269
13:20:45.738823  PSH, ACK - Len: 424 ->            Seq = 3432288489 - 3432288913 (424) Ack = 1629056269
13:20:45.742719  <- ACK - Len: 1360                Seq = 1629056269 - 1629057629 (1360)
13:20:45.742745  <- ACK - Len: 1360                Seq = 1629057629 - 1629058989 (1360)
13:20:45.742756  <- ACK - Len: 1360                Seq = 1629058989 - 1629060349 (1360)
13:20:45.742757  <- PSH, ACK - Len: 278            Seq = 1629060349 - 1629060627 (278) A
13:20:45.742801  <- ACK - Len: 1360                Seq = 1629060627 - 1629061987 (1360)

13:20:46.725114  ACK ->                            Seq = 3432288913 - 3432288913 (0) Ack = 1629056269 Sack: 1629058989 - 1629060349
13:20:46.728406  ACK ->                            Seq = 3432288913 - 3432288913 (0) Ack = 1629056269 Sack: 1629058989 - 1629060627
13:20:46.729048  <- ACK - Len: 1360                Seq = 1629061987 - 1629063347 (1360)
13:20:46.732286  <- ACK - Len: 1360                Seq = 1629063347 - 1629064707 (1360)
13:20:46.741652  ACK ->                            Seq = 3432288913 - 3432288913 (0) Ack = 1629056269 Sack: 1629058989 - 1629061987
13:20:46.745803  <- ACK - Len: 1360                Seq = 1629056269 - 1629057629 (1360)
13:20:46.789870  ACK ->                            Seq = 3432288913 - 3432288913 (0) Ack = 1629056269 Sack: 1629057629 - 1629061987
13:20:46.802818  ACK ->                            Seq = 3432288913 - 3432288913 (0) Ack = 1629061987
13:20:46.806955  <- ACK                            Seq = 1629064707 - 1629064707 (0) Ack

13:20:47.644851  ACK ->                            Seq = 3432288913 - 3432288913 (0) Ack = 1629064707
13:20:47.649395  <- ACK - Len: 1360                Seq = 1629064707 - 1629066067 (1360)
13:20:47.680523  ACK ->                            Seq = 3432288913 - 3432288913 (0) Ack = 1629064707
13:20:47.684308  <- ACK - Len: 1360                Seq = 1629066067 - 1629067427 (1360)
```

Figure 3.6: Wireshark flow graph output of a TCP flow with SACKs

Now we explain in detail each possible case for three candidate reneging instances that look like Misbehavior B instances. Misbehavior B is observed in TCP flows having at least two SACK blocks. Once the data between the ACK and the first SACK are received, a data receiver increases the ACK, but misbehaves and does not acknowledge other SACK blocks. The ack with no SACK blocks implies an instance of reneging.

(I) Figure 3.7 shows a candidate reneging instance where the retransmissions for the data seems to have been reneged are transmitted using multiple data packets. The initial state of the receive buffer is known as: stateACK 92655 stateSACK 93191-93727. The first ack (#2, ACK 92655 SACK 94263-94799 93191-[|tcp]) informs that data (#1) are received out-of-order. The "[|tcp]" indicates that the second SACK block is truncated in the trace file and only the left edge is available to display. A new stateSACK 94263-94799 is added to the state. The next ack (#5, ACK 93068 SACK 93191-94092 94263-[|tcp]) acknowledges the receipt of data packets (#3) and (#4). The stateSACK 93191-93727 is updated to 93191-94092. Data packet (#6, 93068-93604) fills the gap between the ACK and the first SACK block 93191-94092. Consequently, ACK is increased to 94092 (#7) but a SACK for out-of-order data 94263-94799 is not present in the ack (#7) (Misbehavior B). Out-of-order data 94263-94799 are marked as MISSING by RenegDetect v2. Next, two retransmissions are monitored covering the MISSING out-of-order data, (#8, 94092-94628) and (#9, 94628-95164). When the first partial retransmission for the MISSING out-of-order data are received (#8), ACK is increased to 94799 (the right edge of MISSING out-of-order data) instead of 94628 indicating that the MISSING out-of-order data are still in the receive buffer. As a result, we conclude that the candidate reneging instance is a Misbehavior B instance, and not an instance of reneging.

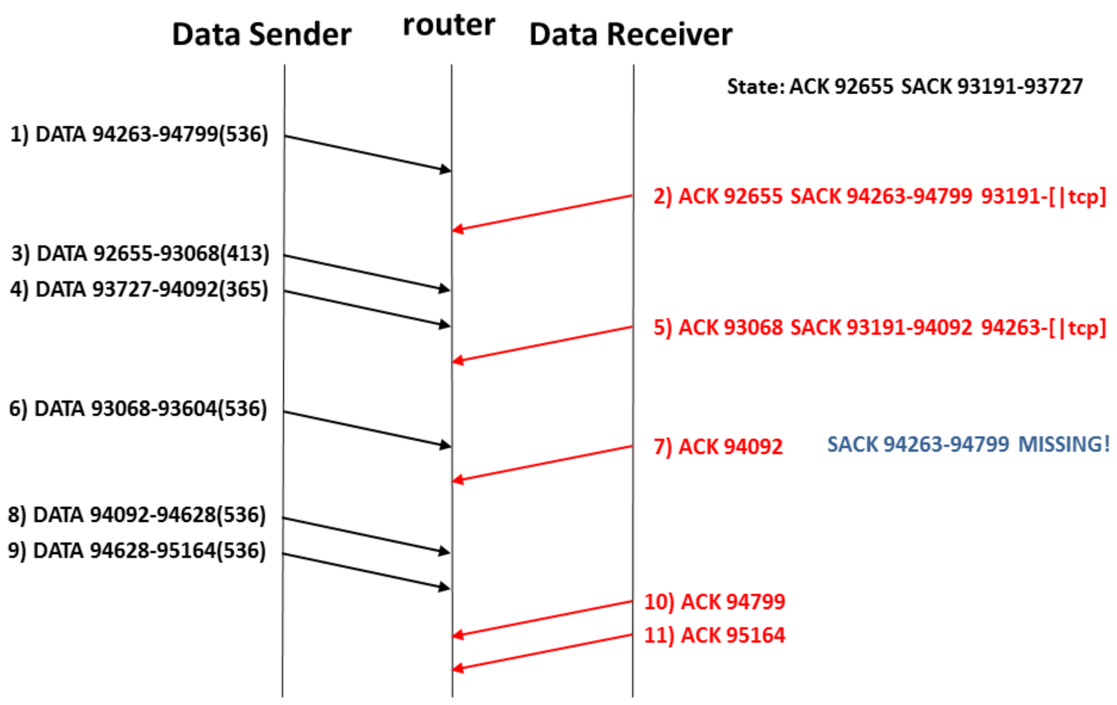


Figure 3.7: Candidate reneging instance I (not reneging)

(II) Figure 3.8 shows a candidate reneging instance where we cannot conclude if the instance is or not reneging. When the data (#9) are received, the ACK is increased to 16850 (#10) but previously SACKed data 18230-19610 are not reported with a SACK block (Misbehavior B). RenegDetect v2 marks bytes 18230-19610 as MISSING. The retransmission for the MISSING bytes is monitored with the next data packet (#11). The reply ack now has a SACK block (18230-19610). Is the new SACK block for the previously received out-of-order data (MISSING) or the retransmission?

We cannot conclude if the observed behavior is an instance of reneging or Misbehavior B due to ambiguity. We report this type of instance as ambiguous.

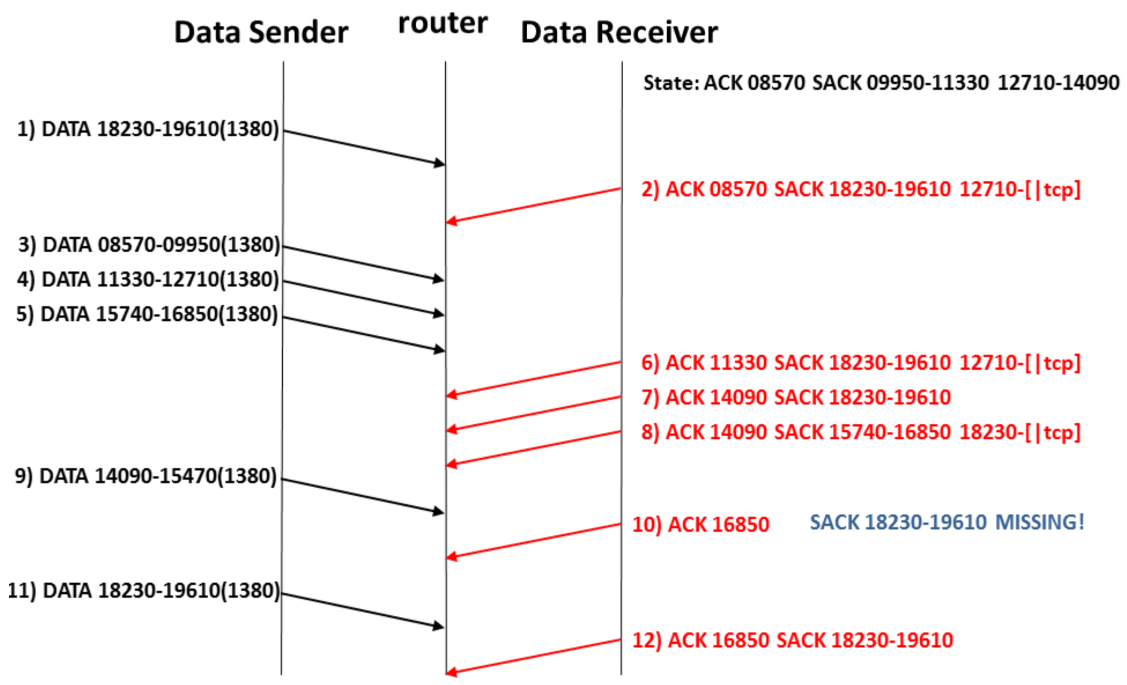

Data Sender
router
Data Receiver
State: ACK 08570 SACK 09950-11330 12710-14090
1) DATA 18230-19610(1380)
2) ACK 08570 SACK 18230-19610 12710-[|tcp]
3) DATA 08570-09950(1380)
4) DATA 11330-12710(1380)
5) DATA 15740-16850(1380)
6) ACK 11330 SACK 18230-19610 12710-[|tcp]
7) ACK 14090 SACK 18230-19610
8) ACK 14090 SACK 15740-16850 18230-[|tcp]
9) DATA 14090-15470(1380)
10) ACK 16850
SACK 18230-19610 MISSING!
11) DATA 18230-19610(1380)
12) ACK 16850 SACK 18230-19610

Figure 3.8: Candidate reneging instance II (ambiguous)

(III) Figure 3.9 shows a candidate reneging instance where the retransmissions for the MISSING out-of-order data are observed and ACK is increased steadily after each retransmission. When data (#8) is received, ACK is increased to 70336 (#9) but no SACKs are reported. RenegDetect v2 marks bytes 70336-74476 as MISSING. Next, the retransmissions for the MISSING out-of-order data are monitored: (#10), (#12), and (#13), respectively. ACK is increased steadily after each retransmission. This behavior clearly indicates a reneging instance.

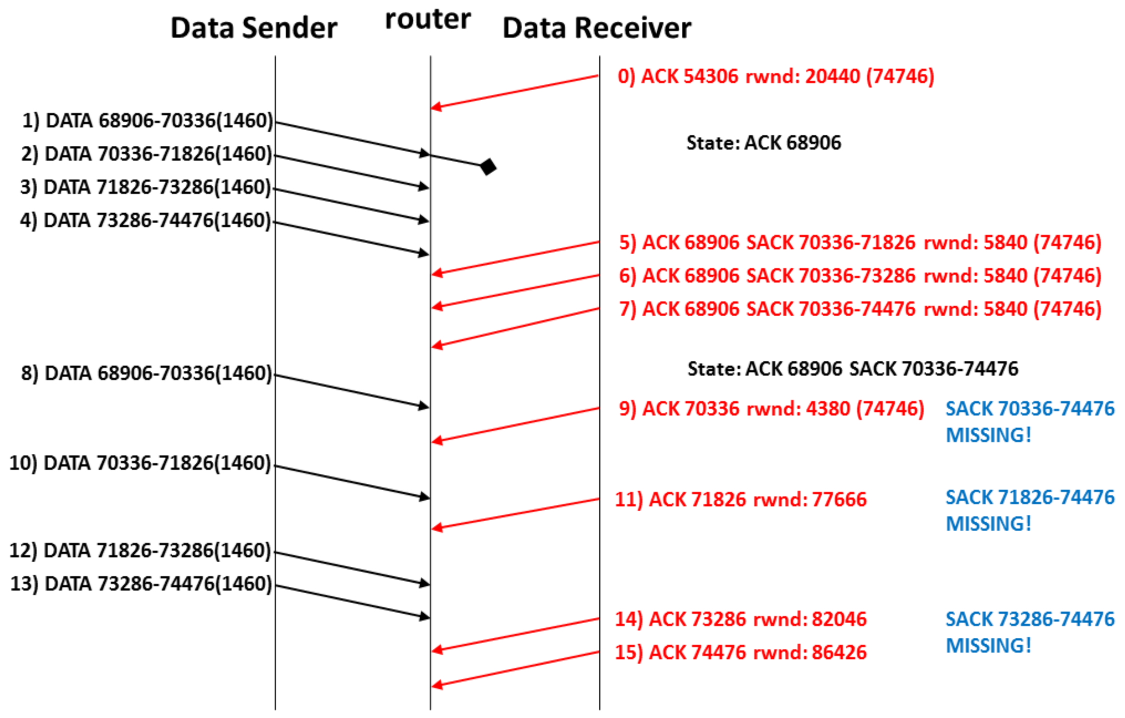

Data Sender
router
Data Receiver
0) ACK 54306 rwnd: 20440 (74746)
1) DATA 68906-70336(1460)
2) DATA 70336-71826(1460)
3) DATA 71826-73286(1460)
4) DATA 73286-74476(1460)
State: ACK 68906
5) ACK 68906 SACK 70336-71826 rwnd: 5840 (74746)
6) ACK 68906 SACK 70336-73286 rwnd: 5840 (74746)
7) ACK 68906 SACK 70336-74476 rwnd: 5840 (74746)
State: ACK 68906 SACK 70336-74476
8) DATA 68906-70336(1460)
9) ACK 70336 rwnd: 4380 (74746)
SACK 70336-74476 MISSING!
10) DATA 70336-71826(1460)
11) ACK 71826 rwnd: 77666
SACK 71826-74476 MISSING!
12) DATA 71826-73286(1460)
13) DATA 73286-74476(1460)
14) ACK 73286 rwnd: 82046
15) ACK 74476 rwnd: 86426
SACK 73286-74476 MISSING!

Figure 3.9: Candidate reneging instance III (reneging)

Each candidate reneging case presented above was analyzed by hand using wireshark. Can we program RenegDetect to identify each case automated? Yes. If we were able to match each data packet to a corresponding ack, RenegDetect could be programmed to identify each case automated. Unfortunately, traces are collected at an intermediate router where data packets can get lost after being monitored and acks can get lost before reaching the router. Such packet losses can cause ambiguity in data-to-ack matching. If traces were collected at the data receiver such an issue would not exist. Each data could be matched to a corresponding ack. Past research, [Jaiswal 2004], identified the same problem where the authors proposed a passive measurement methodology to infer congestion window (cwnd) in traces captured at an intermediate router. We decided to simply analyze each candidate reneging instance using wireshark by hand to avoid programming complexity to implement data-to-ack matching within RenegDetect v2.

The updated RenegDetect, v2, identifies misbehaviors where no or partial data retransmissions are observed. Whenever a misbehavior is observed, the out-of-order data that seem to have been reneged are marked as MISSING and RenegDetect v2 keeps tracks of retransmissions for the MISSING data. If retransmissions are observed for the MISSING data, RenegDetect v2 reports a candidate reneging instance. For each candidate reneging instance, a hand analysis is done using wireshark to determine if the instance is a misbehavior, a reneging, or an ambiguous instance.

Our model cannot detect reneging instances with 100% certainty if particular acks and data PDUs are not observed in a trace. Our model relies on acks to detect inconsistencies between the state of receiver buffer and new SACK information. In addition, our model relies on data retransmissions to distinguish between a reneging and a misbehavior instance. For a reneged flow, if acks that cause inconsistencies were not observed by the intermediate router or lost during the trace capture, reneging would go undetected (the state is still consistent). Similarly, if data retransmissions are not included in the trace capture, reneging again would go undetected. Therefore, the frequency of reneging, *p*%, that we report in our analysis is a lower limit and should be interpreted as "reneging happens in at least *p*% of the TCP flows analyzed".

To report the frequency of reneging, TCP flows monitored at an intermediate router should be analyzed. But, we first needed to determine the minimum number of TCP flows to analyze based on our initial hypothesis that reneging rarely if ever occurs in practice. The following section answers at least how many flows we needed to analyze.

3.2.4 Hypothesis

To generalize reneging behavior, we needed to analyze TCP flows to determine if reneging is happening or not in today's Internet. But how many TCP flows do we need to analyze to be statistically confident of our conclusions?

Given a set of TCP flows, we assumed that whether or not a TCP flow reneges is a binary event with probability P(reneging) = p, and the TCP flows are independent and identically distributed (i.i.d.) with respect to reneging (we discuss if TCP flows form the same host are i.i.d. or not at the end of this section.) We defined event A as reneging happens in a TCP flow. Assuming reneging is a rare event, our initial hypothesis (H_0) was:

$$H_0 \colon p(A) \geq \ 10^{-5}$$

We wanted to design an experiment which rejects H_0 with 95% confidence (confidence coefficient *α*=0.05) thus allowing us to conclude that:

$$p(A) < 10^{-5}$$

Our experiment would analyze *n* TCP flows hoping to not find a single instance of reneging. We wanted to know the value of *n* such that the probability that H_0 is true even though no TCP flow reneges is less than 0.05 (confidence coefficient.)

$$\mathrm{P}(\ \mathrm{k} = 0 \mid H_0\ ) < .05$$

The probability of reneging occurring *k* times in *n* i.i.d. TCP flows is:

$$p_n(k) = \ P\,\{reneging\ occurs\ k\ times\ in\ n\ flows\ in\ any\ order\} = \binom{n}{k} p^k q^{n-k}$$

The probability that reneging does not occur (*k*=0) in *n* trials assuming H_0 is:

$$p_n(0) = \ \binom{n}{0} p^0 q^{n-0}$$

$$p_n(0) = \ (1 - 10^{-5})^n$$

To find the minimum number of TCP flows (n) to analyze, hoping with 95% confidence to reject H_0, we needed:

$$p_n(0) < \alpha$$

$$(1 - 10^{-5})^n < 0.05$$

The minimum n satisfying the equation is 299,572 (derived from MAPLE.)

Now, let us discuss if TCP flows are i.i.d.? To renege, a TCP flow should have out-of-order data in its receive buffer. The out-of-order data exist due to either congestion or packet reordering in the network. Other simultaneous TCP flows from the same host would experience the same congestion or packet reordering if they share the same bottleneck router. Therefore, if one TCP flow reneges, it is expected that other TCP flows from the same TCP host might also renege. For example, FreeBSD employs global reneging (see Section 4.4) where all TCP flows renege simultaneously. On the other hand, Linux and Solaris employ local reneging (see Sections 4.3 and 4.6) where each TCP flow reneges independently. Therefore, some simultaneous TCP flows from same host are i.i.d. and others are not depending on the host's operating system. Initially, we assumed that each TCP flow was independent.

To generalize reneging behavior, our goal was to analyze at least 300K TCP flows with SACKs using RenegDetect v2. If we could document no reneging instances, we could claim that reneging is a rare event, i.e., P(reneging) < 10^{-5}. For that, TCP traces from three domains (Internet, wireless, enterprise) were analyzed using RenegDetect v2. The results of the trace analysis are presented in the next section.

3.2.5 Results of Internet Trace Analysis

In this section, we 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). Our goal was to analyze 300K TCP flows using SACKs and find no instances of reneging. Unfortunately, we found instances of reneging. Therefore, we could not reject our hypothesis H_0 specified in Section 3.2.4 to conclude P(reneging) < 10^{-5}.

Since reneging instances were found, analyzing 300K TCP flows were no longer necessary. As a result, we ended up analyzing 202,877 TCP flows using SACKs from the three domains where a total of 104 instances of reneging were found. The sample proportion of reneging, $\hat{p}$, is

$$\hat{p} = \frac{\mathrm{X}}{n} = \frac{104}{202877} = 0.000512$$

From [Moore 1993], the standard error of sample proportion $\hat{p}$ is

$$SE_{\hat{p}} = \sqrt{\frac{\hat{p}(1-\hat{p})}{n}} = \sqrt{\frac{0.000512(0.999488)}{202877}} = 0.00005$$

An approximate level C confidence interval for an unknown population proportion p is estimated using

$$\hat{p} \ \pm z^{*} SE_{\hat{p}}$$

A 95% confidence interval of event reneging being true for a TCP flow is calculated using z^{*} being 1.960

$$\hat{p} \ \pm z^{*} SE_{\hat{p}} = 0.000512 \ \pm (1.960)(0.00005) = 0.000512 \ \pm \ 0.0000984$$

$$= 0.05\% \ \pm 0.009\%$$

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.

For each reneging instance, we tried to 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 are processed beforehand to have a single trace file for each bidirectional TCP flow using 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.

(1) By documenting reneging instances during the trace analysis, we provide the first set of reneging traces to interested researchers and TCP stack implementors for further analysis. (2) When multiple TCP flows exist in a trace, wireshark views all of them in the same flow graph which makes it hard to read the graph for hand analysis. In addition, if a trace file with multiple flows is large (some of the traces provided by CAIDA are 1-4 GB per trace), wireshark displays an out of memory error and terminates. Therefore, we processed trace files provided by CAIDA, SIGCOMM, and LBNL into separate trace files for each TCP flow using SACKs. Figure 3.10 summarizes the processing of TCP traces.

For the reneging analysis, only TCP flows that contain at least one SACK block during a data transfer are of interest (other flows cannot renege by definition). For each trace file, we first identify TCP flows using SACKs. Flows not using SACKs were discarded. Second, we filter each trace to include only TCP PDUs using *tshark* tool, that is, UDP and ICMP PDUs are discarded. Third, we split the resulting trace into individual TCP PDUs using *editcap* tool. Each individual TCP PDU is named by

4-tuple (source IP, source port, destination IP, destination port) plus a sequence number. Finally, individual PDUs are merged into a single TCP trace using *mergecap* tool for each flow. *Tshark*, *editcap* and *mergecap* are command line utilities provided by wireshark. The process described here corresponds to TCP flow filter box in Figure 3.10.

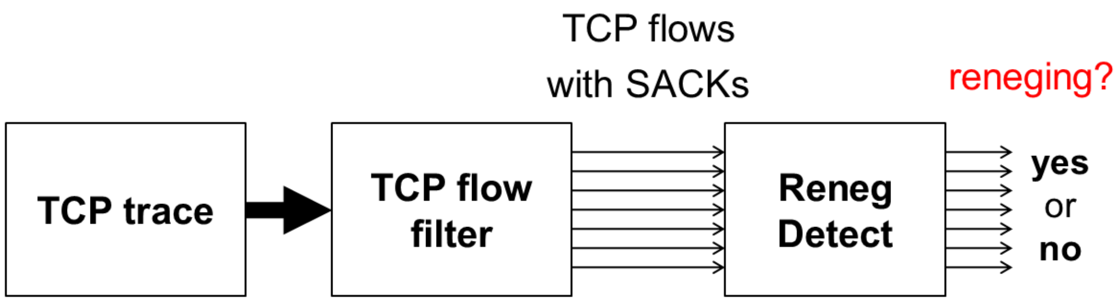


Figure 3.10: Filtering traces

RenegDetect v2 accepts a TCP trace file as an input and analyzes a TCP flow using the model detailed in Sections 3.2.1 and 3.2.3. RenegDetect v2 logs candidate reneging flows (and each individual instance per flow) during the trace analysis. Candidate reneging instances are later inspected by hand using wireshark to conclude reneging or not.

RenegDetect v2 relies on data retransmissions to report candidate reneging instances. For each TCP flow, RenegDetect v2 also keeps track of the percentage of data transmitted by the data sender (data seen between the lowest and the highest ACK) monitored at the intermediate router to check if all data are observed. The initial 5% of the flow is skipped since data sent in prior window may not be available in a TCP trace if the first PDUs in the trace are acks instead of data PDUs. For a data transfer of 100 bytes, for example, if a data collection point observes the last 95 bytes on the data path, we classify that flow as 100% of the data is observed. If all the data is observed at the router, we can argue that retransmissions follow the same data path, which gives us a strong argument to rely on retransmissions to infer reneging. If the routing path changes during a data transfer or large number of packets are dropped during a trace capture, then a gap in the data will be observed (so the data percentage would be less than 100%). If RenegDetect v2 observes, for instance, only 40% of the data, then we can argue that relying on the retransmissions to detect reneging instances is unreliable since some of the data and retransmissions (the other 60% of the data) followed a different path or got lost during trace capture. If we analyze incomplete flows having gaps in the data, less reneging instances would be detected. For such flows, reneging instances could be identified falsely as non-reneging since data retransmissions might be missing. To avoid a bias to identify less reneging instances, we only analyze TCP flows where at least 95% of the data is available and ignore the rest.

In all traces provided (CAIDA, SIGCOMM, and LBNL), the real IP addresses of data senders/receivers were remapped to other IP addresses for privacy and security purposes. This process is called IP anonymization.

The outline of the rest of this section is as follows. The frequency of reneging for Internet (CAIDA), wireless (SIGCOMM), enterprise (LBNL) domains are presented in Sections 3.2.5.1, 3.2.5.2, and 3.2.5.3, respectively.

3.2.5.1 Cooperative Association of Internet Data Analysis (CAIDA) Traces

In this section, we present the reneging frequency of Internet backbone traces captured between April 2008 and October 2010. A brief description of the traces is presented in Section 3.2.5.1.1. The frequency of reneging in CAIDA traces is reported in Section 3.2.5.1.2. During the trace analysis, 104 reneging instances were detected. Each reneging instance was analyzed in detail and categorized based on the operating system guess of the data receiver. We detail reneging instances for Linux, FreeBSD, and Windows operating systems in Sections 3.2.5.1.3, 3.2.5.1.4, and 3.2.5.1.5, respectively.

3.2.5.1.1 Description of Traces

The trace files from CAIDA [Caida] are representative of a wide area Internet traffic and were collected via data collection monitors set in Equinix data centers (http://www.equinix.com) in Chicago and San Jose, CA. The monitors are set on OC-192 Internet bi-directional backbone links (9953 Mbps) between (Seattle and Chicago) and (San Jose and Los Angeles). The trace data were collected separately for each direction of the bi-directional links: direction A (Seattle to Chicago, San Jose to Los Angeles) and direction B (Chicago to Seattle, Los Angeles to San Jose) and the traces for each direction are provided in separate files.

The actual amount of data captured from each frame (snaplen) by the monitors was 48 bytes as opposed to 68 bytes which is default by tcpdump [Tcpdump]. This 48 byte limit causes some SACK blocks to be truncated whenever a TCP timestamp option is present or there are multiple SACK blocks. If all SACKs of a TCP flow are truncated, the flow is discarded.

CAIDA provides 60 minute long traces for each Equinix monitor (Chicago, San Jose) per month. In our lab, we did not have enough processing capacity to filter all CAIDA traces as described in Section 3.2.5. Instead, we processed 2 minute long traces for each month whenever trace data was available for both directions. When we processed all the 2 minute traces from April, 2008 to October, 2010 for both monitors, we processed another set of 2 minute traces for Chicago monitor from April, 2008 to January, 2009. For some dates, traces were only available for one direction (especially for San Jose monitor). In such case, either data or ack traffic was available but not both. We ignored those traces in our analysis. When we detected reneging instances, we also filtered 10 minute long traces (covering the 2 minute trace) for the reneged data receivers to analyze reneging behavior for longer durations and more detail.

Tables 3.2 and 3.3 show the statistics for the percentage of data of the TCP flows using SACKs in CAIDA traces captured at Chicago monitor for directions A and B, respectively. Tables 3.4 and 3.5 show the same statistics for the San Jose monitor. The first six columns of the table show the date (yyyymmdd), the minute interval analyzed, and percentage of TCP flows where 100, [95, 100), (0, 95), and 0 percent of the data is observed at the intermediate router. The seventh column shows the percentage of TCP flows where there were multiple TCP flows for the same 4-tuple in the trace. We identified multiple TCP flows using *tcptrace*, a TCP connection analysis tool. RenegDetect v2 is designed to analyze a single bidirectional TCP flow and does not have the ability to distinguish which data/acks belong to which flow when multiple TCP flows exist in a trace file. Such ability was not implemented not to increase the programming complexity of RenegDetect v2. That is, RenegDetect v2 cannot operate when multiple TCP flows exist in the same trace. Thus, trace files with multiple TCP flows were ignored (column 7) along with those flows where less than 95% of the data was available (columns 5 and 6).

In Table 3.2, on dates 20080619 and 20080717 (the rows highlighted with grey color), the percentages of data that falls between (0, 95) interval are 41.63% and

66.10%, respectively for the Chicago monitor (direction A). This behavior implies that gaps in the data were observed due to a route change or high packet losses during trace capture. We ignored these TCP traces since missing data/retransmissions would bias the results in favor of not reneging instances.

Table 3.2: Percentage of data monitored in CAIDA traces (Chicago, direction A)

Date	Minutes	100%	[95, 100)%	(0, 95)%	0%	Multiple Flows
20080430	20-22	96.31	2.33	0.64	0.64	0.09
20080430	31-33	96.00	2.22	0.66	0.63	0.50
20080430	20-31	75.02	2.53	0.00	7.51	14.93
20080515	57-59	84.02	14.73	0.55	0.60	0.10
20080515	37-39	83.07	15.39	0.65	0.81	0.08
20080515	50-60	64.00	28.00	0.00	7.64	0.36
20080619	00-21	48.00	6.61	41.63	1.77	1.99
20080717	11-13	26.28	3.81	66.10	3.55	0.26
20080821	18-20	95.27	1.65	1.13	1.75	0.21
20080821	37-39	94.75	2.25	1.12	1.69	0.19
20080918	03-05	96.28	1.28	0.81	1.51	0.12
20080918	02-04	95.72	1.71	0.73	1.83	0.00
20081016	11-13	94.01	1.66	1.91	2.41	0.00
20081016	23-25	93.25	2.08	1.21	3.29	0.17
20081120	52-54	95.06	1.49	1.49	1.72	0.23
20081120	25-27	95.77	1.18	0.59	2.12	0.35
20081218	45-47	91.65	2.82	2.82	2.09	0.63
20081218	09-11	95.88	2.11	1.11	0.91	0.00
20090115	18-20	90.53	6.63	0.90	1.44	0.50
20090115	53-55	89.46	5.79	1.41	2.92	0.41
20090219	48-50	94.83	0.25	2.22	1.97	0.74
20090331	23-25	96.84	1.16	0.95	0.53	0.53
20090416	28-30	95.74	1.84	1.27	1.15	0.00
20090521	10-12	91.01	4.03	2.33	2.17	0.47
20090618	34-36	92.21	2.79	2.35	2.50	0.15
20090716	22-24	90.75	6.33	0.97	1.95	0.00
20090820	32-34	97.81	0.73	0.73	0.73	0.00

Table 3.2 continued

20090917	41-43	95.86	1.48	1.18	0.59	0.89
20091015	17-19	92.48	1.57	2.51	1.57	1.88
20091119	13-15	80.51	5.32	2.03	8.61	3.54
20091217	06-08	87.32	6.83	4.88	0.00	0.98
20100121	48-50	93.98	2.52	2.63	0.55	0.33
20100225	45-47	97.04	0.59	1.18	0.99	0.20
20100325	23-25	96.25	2.50	0.00	1.25	0.00
20100414	10-12	84.85	9.09	0.00	6.06	0.00
20100819	44-46	94.13	1.28	1.28	3.06	0.26
20100916	15-17	90.28	4.17	1.39	3.47	0.69
20101029	42-44	86.49	6.49	2.16	4.86	0.00

Table 3.3: Percentage of data monitored in CAIDA traces (Chicago, direction B)

Date	Minutes	100%	[95, 100)%	(0, 95)%	0%	Multiple Flows
20080430	20-22	94.52	0.07	0.85	4.57	0.00
20080430	31-33	96.57	0.50	0.50	2.37	0.06
20080515	57-59	97.24	0.09	0.31	2.28	0.09
20080515	37-39	97.44	0.20	0.20	2.16	0.00
20080619	00-21	93.63	0.07	0.11	2.45	3.75
20080717	11-13	92.01	0.31	0.61	6.62	0.46
20080821	18-20	98.37	0.57	0.49	0.46	0.11
20080821	37-39	97.94	0.38	0.70	0.91	0.07
20080918	03-05	97.41	0.65	0.88	0.97	0.09
20080918	02-04	97.11	0.66	0.95	1.18	0.09
20081016	11-13	96.97	0.57	1.39	0.88	0.19
20081016	23-25	97.72	0.40	0.89	0.92	0.06
20081120	52-54	96.99	0.94	0.47	1.42	0.18
20081120	25-27	96.40	0.35	0.83	1.94	0.48
20081218	45-47	96.10	0.53	0.53	2.60	0.24
20081218	09-11	96.60	0.63	1.42	1.34	0.00
20090115	18-20	95.90	0.88	0.67	2.13	0.41
20090115	53-55	95.24	1.40	0.75	2.00	0.60
20090115	14-24	99.22	0.00	0.00	0.00	0.78

Table 3.3 continued

20090219	48-50	93.24	1.29	1.13	3.54	0.81
20090331	23-25	97.82	0.38	0.64	1.02	0.13
20090416	28-30	97.47	0.78	0.54	1.14	0.06
20090521	10-12	98.57	0.07	0.14	1.22	0.00
20090618	34-36	94.50	0.59	0.39	4.38	0.13
20090716	22-24	87.90	0.36	0.72	11.02	0.00
20090820	32-34	97.85	0.86	0.43	0.86	0.00
20090917	41-43	95.12	0.70	1.39	2.44	0.35
20091015	17-19	96.19	0.21	0.64	2.97	0.00
20091119	13-15	95.04	0.00	0.76	4.20	0.00
20091217	06-08	96.33	0.00	0.00	3.67	0.00
20100121	48-50	88.82	1.64	0.00	9.21	0.33
20100225	45-47	95.45	0.00	1.01	3.54	0.00
20100325	23-25	90.30	0.75	8.21	0.00	0.75
20100414	10-12	91.67	2.08	6.25	0.00	0.00
20100819	44-46	93.29	0.67	0.00	5.37	0.67
20100916	15-17	92.97	0.00	0.00	5.41	1.62
20101029	42-44	95.02	0.90	0.45	3.62	0.00

Table 3.4: Percentage of data monitored in CAIDA traces (San Jose, direction A)

Date	Minutes	100%	[95, 100)%	(0, 95)%	0%	Multiple Flows
20080717	06-08	97.30	0.58	0.97	1.09	0.06
20080821	24-26	95.92	0.39	1.12	2.35	0.22
20081016	03-05	98.10	0.40	0.70	0.62	0.17
20081120	34-36	95.72	0.82	1.76	1.59	0.11
20090331	28-30	96.85	0.79	1.33	0.75	0.27
20090416	14-16	92.44	5.89	0.92	0.63	0.13
20090521	25-27	94.61	1.41	2.28	1.07	0.63
20090618	03-05	95.56	1.19	1.86	1.23	0.16
20090716	40-42	95.31	1.72	2.15	0.77	0.05
20090820	11-13	97.85	0.40	1.07	0.54	0.13
20090820	07-17	87.67	0.00	0.00	12.33	0.00
20090917	20-22	95.72	1.06	2.24	0.86	0.12

Table 3.4 continued

20091015	21-23	95.13	1.12	2.00	1.33	0.41
20091119	12-14	96.92	0.86	1.27	0.82	0.14
20091217	04-06	97.03	0.27	1.35	0.81	0.54
20100121	55-57	95.69	0.77	1.00	2.54	0.00
20100225	53-55	96.91	0.39	0.19	2.32	0.19
20100414	58-60	93.04	3.78	0.85	2.19	0.14

Table 3.5: Percentage of data monitored in CAIDA traces (San Jose, direction B)

Date	Minutes	100%	[95, 100)%	(0, 95)%	0%	Multiple Flows
20080717	06-08	95.45	0.91	2.07	1.49	0.08
20080821	24-26	96.29	0.40	1.72	1.59	0.00
20081016	03-05	95.22	0.57	1.24	2.77	0.19
20081120	34-36	94.80	1.44	1.52	2.09	0.14
20090331	28-30	93.80	0.76	0.90	4.20	0.34
20090331	24-34	0.00	100.00	0.00	0.00	0.00
20090416	14-16	96.28	0.83	0.83	1.83	0.24
20090521	25-27	94.18	0.52	0.26	4.58	0.46
20090618	03-05	93.08	0.65	1.38	4.72	0.16
20090716	40-42	94.30	0.31	0.78	4.14	0.47
20090820	11-13	98.33	0.33	0.33	0.83	0.17
20090917	20-22	96.30	0.21	0.58	2.80	0.11
20091015	21-23	93.37	0.73	0.70	4.75	0.45
20091119	12-14	94.79	0.74	0.65	3.35	0.47
20091217	04-06	94.34	0.00	1.74	3.27	0.65
20100121	55-57	96.09	0.95	0.59	2.37	0.00
20100225	53-55	97.53	0.82	0.59	1.06	0.00
20100414	58-60	98.24	0.19	0.33	1.19	0.05

3.2.5.1.2 Results

In this section, we document the frequency of reneging in the CAIDA traces. Table 3.6 presents the number of TCP flows using SACKs and the frequency of reneging. The columns of the table show the date (in yyyymmdd format), the number of TCP flows using SACK blocks analyzed, the number of candidate reneging flows, and the number of reneged flows, respectively. The candidate reneging flows are those that satisfy the following two conditions: (a) some SACK block(s) were MISSING and (b) data retransmissions for the MISSING SACK block(s) were observed. Each candidate reneging flow was analyzed by hand using the wireshark tool [Wireshark] to determine if reneging happened or if the candidate reneging instance was an instance of misbehavior. For each date, we report an aggregate amount of TCP flows using SACKs from two the monitors: Chicago and San Jose. We report that out of 1273 candidate reneging flows (0.78% of all flows) analyzed, 104 flows (0.05%) reneged.

We analyzed each reneging flow in detail and categorized reneging instances based on the operating system of the data receiver. We detail reneging instances and behavior for Linux, FreeBSD, and Windows operating systems in Sections 3.2.5.1.3, 3.2.5.1.4, and 3.2.5.1.5, respectively.

Table 3.6: Reneging frequency of CAIDA traces

Date	Flows using SACKs	Candidate Reneging Flows	Reneged Flows
20080430	10434	97	8
20080515	12233	111	40
20080619	22377	85	1
20080717	4507	40	0
20080821	10797	64	0
20080918	5835	32	0
20081016	13493	66	1
20081120	7829	73	0
20081218	4755	42	1
20090115	7998	130	11
20090219	977	15	0
20090331	6699	50	1
20090416	6484	53	0
20090521	5440	34	0

Table 3.6 continued

20090618	5700	60	1
20090716	4379	67	0
20090820	1755	46	40
20090917	4901	42	0
20091015	7901	68	0
20091119	3773	27	0
20091217	1690	17	0
20100121	2848	23	0
20100225	2022	9	0
20100325	201	2	0
20100414	4896	12	0
20100819	514	5	0
20100916	446	1	0
20101029	556	2	0
TOTAL	161440	1273	104

3.2.5.1.3 Linux Reneging Instances

In this section, we characterize Linux reneging instances detected in the CAIDA trace analysis. First, Linux data receivers are inferred by examining TCP fingerprints of the reneging flows. Second, detailed statistics regarding the frequency of reneging are listed. Next, for the reneging data receivers, we analyze the characteristics of the reneging and non-reneging connections and the type of reneging employed (global vs. local). Then, a typical Linux reneging instance is presented. Finally, circumstances of Linux reneging are presented.

We strongly believe that reneging data receivers listed in Table 3.7 were running Linux. Table 3.7 details the TCP fingerprints (characteristics) of the reneging data receivers. The columns of the table show the date (in yyyymmdd format), the anonymized IP address of the reneging data receiver, maximum segment size (MSS), window scale value, initial receiver window (rwnd), maximum rwnd value observed during the connection, if timestamps (TS) [RFC1323] were used, and if DSACKs

[RFC2883] were used, respectively. We believe these data receivers were running Linux since they all exhibited the following behaviors. (1) Linux implements dynamic right-sizing (DRS) where the rwnd dynamically changes based on the receiver's estimate of the sender's congestion window [Fisk 2001]. With DRS, the initial advertised rwnd of a Linux TCP is 5840 bytes (column 5) and changes dynamically (column 6) over the course of the connection. (2) Linux TCP supports DSACKs by default (sysctl *net.ipv4.tcp_dsack* = 1) and DSACKs [RFC2883] were observed for all the data receivers (column 7).

Table 3.7: Host characteristics for Linux data receivers

Date	Anonymized IP	MSS (SYN)	Win Scale	Rwnd (SYN)	Rwnd (Max)	TS	DSACK
20080430	226.186.117.234	1460	n/a	5840	auto	no	yes
20080430	226.186.117.238	1460	n/a	5840	auto	no	yes
20080515	226.186.116.218	1460	n/a	5840	auto	no	yes
20080515	226.186.116.219	1460	n/a	5840	auto	no	yes
20080515	226.186.116.221	1460	n/a	5840	auto	no	yes

Table 3.8: Linux reneging instances

Date	Anonymized IP	Reneged Flows	Total Reneging Instances	Total Reneged Bytes	Avg. Reneged Bytes
20080430	226.186.117.234	4	9	24820	2758
20080430	226.186.117.238	2	3	24820	8273
20080515	226.186.116.218	28	74	146000	1973
20080515	226.186.116.219	4	25	102200	4088
20080515	226.186.116.221	2	3	11680	3893
TOTAL		40	114	309520	2715

Table 3.8 reports the reneging instances detected at the Linux data receivers. The columns of the table show the date, the anonymized IP address of the reneged data receiver, the number of TCP flows (connections) reneged, the total number of reneging instances observed in the reneged flows, total amount of reneged bytes, and the average amount of bytes reneged per reneging instance, respectively. A total of 114 reneging instances were observed in 40 flows from 5 different Linux data receivers. The observation suggests that multiple TCP flows renege per each data receiver. The average number of reneging instances per flow was 2.85 (114/40) which indicates that reneging occurs multiple times per TCP flow. The average amount of bytes reneged per reneging instance was 2715 bytes (~2 MSS PDUs.) We report the average amount of bytes reneged per reneging instance to check if reneging occurs when a significant portion of the receive buffer is filled with out-of-order data.

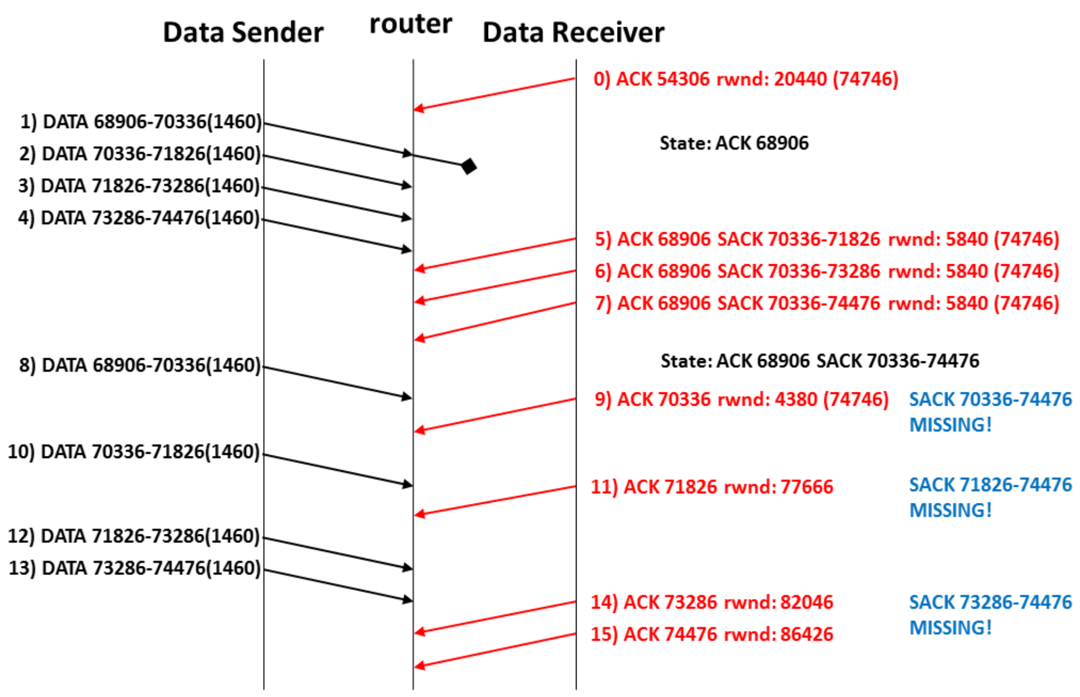
To identify if reneging Linux data receivers were busy servers or clients having large number of TCP flows, we counted the number of TCP connections analyzed for each reneging data receiver. Table 3.9 reports the number of flows using SACKs for the reneging Linux data receivers for the three traces analyzed. 2 minute traces only contain flows using SACKs. In addition to flows using SACKs, the 10 minute trace was filtered to contain TCP flows not using SACKs to infer connection characteristics of a reneging data receiver. In Table 3.9, the number of connections indicates that reneging happens at Linux data receivers having hundreds of TCP flows. We initially expected reneging to happen at busy servers (e.g., web, mail servers) with large number of TCP connections established. In our analysis, all the reneging Linux data receivers were busy clients transferring data from web servers instead.

Table 3.9: Connection numbers for reneging Linux data receivers

Date	Anonymized IP	Flows using SACKs			Flows not using SACKs
		2 minute	2 minute	10 minute	10 minute
20080430	226.186.117.234	105	120	452	5846
20080430	226.186.117.238	147	134	618	3356
20080515	226.186.116.218	107	63		
20080515	226.186.116.219	76	106		
20080515	226.186.116.221	74	107		

Linux employs local reneging where simultaneous TCP connections renege independently (see Section 4.3.1). To confirm local reneging, we analyzed reneging times for each data receiver and verified that reneging instances from different flows occurred at different times independently.

Now let us detail a Linux reneging instance, shown in Figure 3.11, observed at 2008/04/30 on a data receiver identified with anonymized IP 226.186.117.234. The initial state of the receive buffer is as follows: stateACK=68906 with no stateSACKs. First, data packet (#1, 68906-70336), is monitored at the intermediate router. Next, data packets (#2, 70336-71826), (#3, 71826-73286), and (#4, 73286-74476) are observed. The data receiver acknowledges the receipt of out-of-order data packet (#2) with an ack (#5, ACK 68906 SACK 70336-71826). Similarly, when data packets (#3) and (#4) are received out of order, reply acks (#6, ACK 68906 SACK 70336-73286) and (#7, ACK 68906 SACK 70336-74476) are sent to the data sender. Those acks give the impression that data packet (#1) is lost in the network. The state is updated when acks (#5), (#6), and (#7) are each monitored at the router. The state becomes: stateACK 68906 and stateSACK: 70336-74476. Next, a fast-retransmission (#8, 68906-70336) for the data packet (#1) is observed. The data receiver replies with an ack (#9, ACK 70336). When ack (#9) is compared with the state, an inconsistency exists. Previously SACKed data 70336-74476 is missing in the ack (#9). At that point, RenegDetect v2 marks those bytes as MISSING and checks if those bytes are retransmitted. Next, a retransmission (#10, 70336-71826) is monitored for the data packet (#2). The reply ack (#11, ACK 71826) indicates that retransmission (#10) is received in order. Similarly, retransmissions (#12, 71826-73286), (#13, 73286-74476) are observed for reneged data packets (#3) and (#4). After each retransmission, ACK is increased steadily. Therefore, we conclude that reneging occurred.



Data Sender
router
Data Receiver
0) ACK 54306 rwnd: 20440 (74746)
1) DATA 68906-70336(1460)
2) DATA 70336-71826(1460)
3) DATA 71826-73286(1460)
4) DATA 73286-74476(1460)
State: ACK 68906
5) ACK 68906 SACK 70336-71826 rwnd: 5840 (74746)
6) ACK 68906 SACK 70336-73286 rwnd: 5840 (74746)
7) ACK 68906 SACK 70336-74476 rwnd: 5840 (74746)
State: ACK 68906 SACK 70336-74476
8) DATA 68906-70336(1460)
9) ACK 70336 rwnd: 4380 (74746)
SACK 70336-74476 MISSING!
10) DATA 70336-71826(1460)
11) ACK 71826 rwnd: 77666
SACK 71826-74476 MISSING!
12) DATA 71826-73286(1460)
13) DATA 73286-74476(1460)
14) ACK 73286 rwnd: 82046
15) ACK 74476 rwnd: 86426
SACK 73286-74476 MISSING!

Figure 3.11: A Linux reneging instance

In [Seth 2008], 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. In Figure 3.11, the ack (#0, ACK 54306 rwnd: 20440) specifies the rwnd to be (54306-74746). The ack (#7, ACK 68906 SACK 70336-74476 rwnd: 4380) indicates that 14600 bytes (the difference between the ACK values) are received in order since the receipt of ack (#0). In ack (#7), the rwnd still has the same right edge (74746), meaning that 14600 bytes in-order data are not yet read by the receiving application and still reside in the receive buffer satisfying (a). For all reneging Linux TCP flows, this same behavior was exhibited; the advertised rwnd fluctuated and usually became 0 since in-order data were not immediately read by the receiving application. When we investigated non-reneging TCP flows, in general, the rwnd did not fluctuate, meaning that the receiving applications were reading the in-order data immediately. As a result, we confirm that Linux reneges when a receiving application is unable to read in-order data.

According to [Seth 2008], reneging in Linux is expected to happen when a large number of out-of-order segments sit in the receive buffer. Unfortunately, our analysis showed that the average amount of bytes reneged per reneging instance was 2715 bytes (~2 MSS PDUs.) This average amount of reneged out-of-order data does not seem large when compared to Linux's 87380 byte default receive buffer size (sysctl *net.ipv4.tcp_rmem* = 4096 (min) 87380 (default) 2605056(max)). On average, ~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.

3.2.5.1.4 FreeBSD Reneging Instances

This section reports FreeBSD reneging instances observed in the CAIDA traces. First, we explain how we inferred that data receivers were running FreeBSD by examining TCP fingerprints of the reneging flows. Second, detailed statistics regarding the frequency of reneging are listed. Next, for the reneging data receivers, we analyze the characteristics of the reneging and non-reneging connections and type of reneging employed (global vs. local). We conclude by presenting a detailed FreeBSD reneging instance.

We believe that the reneging data receivers listed in Table 3.10 were running FreeBSD. Table 3.10 presents the TCP fingerprints of the reneging hosts. Both reneging hosts had an initial rwnd of 65535 and used timestamps [RFC1323]. Table 3.11 lists the initial rwnd reported in SYN segments of various operating systems observed during our TBIT testing in Chapter 2. 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. Windows 2000 reports at most 2 SACK blocks (Misbehavior A2, see Section 2.4) in a TCP PDU. 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 the reneging data receivers were running FreeBSD.

Table 3.12 reports the frequency of reneging for the FreeBSD data receivers. A total of 11 reneging instances were observed in 11 flows from 2 different hosts. For each flow reneged, a single reneging instance was observed. The average bytes reneged per reneging instance was 3717 bytes (~2.5 MSS PDUs.) This average amount of reneged out-of-order data does not seem significant when compared to FreeBSD's 65535 byte default receive buffer size (sysctl *net.inet.tcp.recvspace: 65536*). On average, ~5.6% of the receiver buffer was allocated to reneged out-of-order data. This behavior indicates that FreeBSD reneges irrespective of out-of-order data size.

Table 3.10: Host characteristics for FreeBSD data receivers

Date	Anonymized IP	MSS (SYN)	Win Scale	Rwnd (SYN)	Rwnd (Max)	TS	DSACK
20081218	238.47.123.36	1460	1	65535	65535	yes	no
20090115	47.179.43.28	1460	1	65535	65535	yes	no

Table 3.11: Initial advertised rwnd (SYN segments) of various operating systems

Operating System	Default Receive Buffer (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 3.12: FreeBSD reneging instances

Date	Anonymized IP	Reneged Flows	Total Reneging Instances	Total Reneged Bytes	Avg. Reneged Bytes
20081218	238.47.123.36	1	1	4380	4380
20090115	47.179.43.28	10	10	36500	3650
TOTAL		11	11	40880	3716

Table 3.13: Connection numbers for FreeBSD data receivers

Date	Anonymized IP	Flows using SACKs			Flows not using SACKs
		2 minute	2 minute	10 minute	10 minute
20081218	238.47.123.36	1	9		
20090115	47.179.43.28	58	5	127	14

To check if reneging FreeBSD data receivers were busy servers or clients, we counted the number of TCP connections analyzed for each reneging data receiver. Table 3.13 presents the results. For FreeBSD, the reneging data receivers did not seem busy. We admit that other TCP flows could be established to the reneging data receivers which were not available in our traces. As with the Linux reneging data receivers did, all of the reneging FreeBSD data receivers (clients) were transferring data from web servers.

FreeBSD employs global reneging where all TCP connections having out-of-order data are reneged simultaneously (see Section 4.4). To confirm this behavior, we analyzed reneging times for the data receiver identified with IP 47.179.43.28 on 2009/01/15. Table 3.14 reports the timestamp values for two acks observed at the intermediate router. The first timestamp (column 4) is for the last ack monitored where the comparison with the receive buffer state was still consistent. The next timestamp (column 5) is for the ack that caused detecting of the reneging instance. Reneging is presumed to have occurred sometime between those two timestamps. The timestamps are clustered around two values: 09:19:02.0xx (flows with port numbers: 50490, 55470, and 61942) and 09:19:31.5yy (flows with port numbers: 50265, 54867, 56888, and 62318). These clustered timestamps indicate that global reneging is employed.

Table 3.14: Timestamp values of a reneging FreeBSD data receiver

Date	Anonymized IP	Port	Timestamp of ack Before Reneging	Timestamp of ack Detecting Reneging
20090115	47.179.43.28	50265	09:19:31.589	09:19:31.735
20090115	47.179.43.28	50490	09:19:02.085	09:19:02.131
20090115	47.179.43.28	54867	09:19:31.584	09:19:31.631
20090115	47.179.43.28	55470	09:19:02.106	09:19:02.153
20090115	47.179.43.28	56888	09:19:31.567	09:19:31.713
20090115	47.179.43.28	59319	09:15:54.138	09:15:54.285
20090115	47.179.43.28	61942	09:19:02.060	09:19:02.405
20090115	47.179.43.28	62318	09:19:31.571	09:19:31.617
20090115	47.179.43.28	63763	09:19:12.112	09:19:12.449
20090115	47.179.43.28	64543	09:19:31.600	09:19:31.647

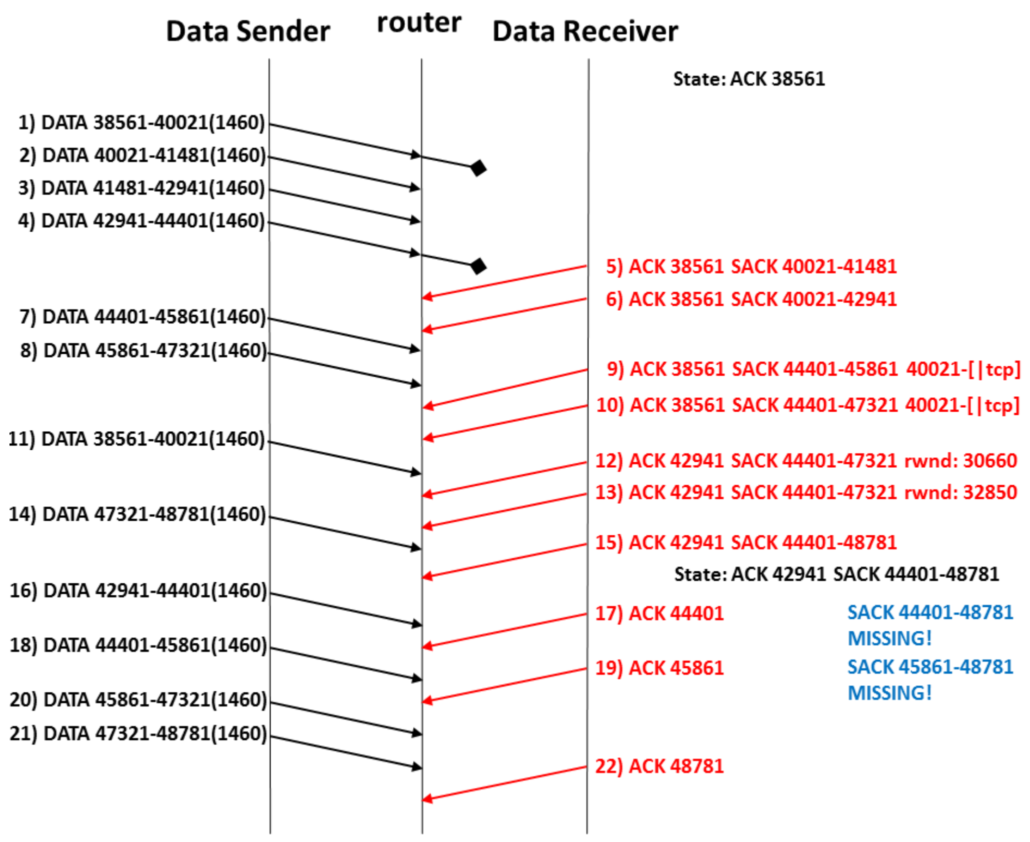

Data Sender
router
Data Receiver
State: ACK 38561
1) DATA 38561-40021(1460)
2) DATA 40021-41481(1460)
3) DATA 41481-42941(1460)
4) DATA 42941-44401(1460)
5) ACK 38561 SACK 40021-41481
6) ACK 38561 SACK 40021-42941
7) DATA 44401-45861(1460)
8) DATA 45861-47321(1460)
9) ACK 38561 SACK 44401-45861 40021-[|tcp]
10) ACK 38561 SACK 44401-47321 40021-[|tcp]
11) DATA 38561-40021(1460)
12) ACK 42941 SACK 44401-47321 rwnd: 30660
13) ACK 42941 SACK 44401-47321 rwnd: 32850
14) DATA 47321-48781(1460)
15) ACK 42941 SACK 44401-48781
State: ACK 42941 SACK 44401-48781
16) DATA 42941-44401(1460)
17) ACK 44401
SACK 44401-48781 MISSING!
18) DATA 44401-45861(1460)
19) ACK 45861
SACK 45861-48781 MISSING!
20) DATA 45861-47321(1460)
21) DATA 47321-48781(1460)
22) ACK 48781

Figure 3.12: A FreeBSD reneging instance

Figure 3.12 shows a FreeBSD reneging instance on 2009/01/15. Reneging is detected with the receipt of ack (#17, ACK 44401) which informs that previously SACKed data 44401-48781 are MISSING. Later, retransmissions for the MISSING data are monitored (data packets (#18, #20, and #21) and ACK is increased steadily after each retransmission. Therefore, we conclude that reneging occurred.

3.2.5.1.5 Windows Reneging Instances

This section characterizes Windows reneging instances observed in the CAIDA traces. First, we explain how we inferred that data receivers were running Windows by examining reneged flows TCP fingerprints. Second, detailed statistics regarding the frequency of reneging are reported. Next, for the reneging data receivers, we analyze the characteristics of the reneging and non-reneging connections and type of reneging employed (global vs. local). Finally, a representative Windows reneging instance is detailed.

We believe that reneging data receivers listed in Table 3.15 are Windows hosts. Table 3.15 details the fingerprints (characteristics) of the reneging hosts. The reneging data receivers did not use the window scale, timestamp, and DSACK options. In addition, all of the reneging data receivers reported at most 2 SACK blocks and the data receivers identified with IPs: 45.36.231.185 and 247.9.212.28 reported at most 2 SACKs when it was known that at least 3 SACK blocks existed at the receiver (Misbehavior A2). Misbehavior A2 is observed only in Windows 2000, XP and Server 2003 (see Section 2.4). These TCP fingerprints suggest that the reneging data receivers were running Windows 2000, XP or Server 2003. The TCP/IP implementation for those operating systems is detailed in [MacDonald 2000] and [Windows 2003]. First, all three operating systems support the window scale and timestamp option [RFC1323]. By default, a Windows host does not advertise these options but enables their use if the TCP peer that is initiating communication includes them in the SYN segment. Second, for the three Windows systems, the advertised rwnd is determined based on the media speed. [Windows 2003] 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. The data receivers specified with IPs: 7.30.83.155 and 126.14.171.216 did not match this specification. But their maximum rwnd was set to 25*MSS and 45*MSS during the course of connection, respectively. Both [MacDonald 2000] and [Windows 2003] specify that Windows TCP adjusts rwnd to even increments of the maximum segment size (MSS) negotiated during connection setup. This specification makes us believe those data receivers were running Windows.

Table 3.15: Host characteristics for Windows data receivers

Date	Anonymized IP	MSS (SYN)	Win Scale	Rwnd (SYN)	Rwnd (Max)	TS	DSACK
20080430	59.190.212.36	1452	n/a	16384	17424	no	no
20080430	247.9.212.28	n/a	n/a	n/a	61320	no	no
20080515	7.30.83.155	1360	n/a	32767	34000	no	no
20080619	238.20.116.194	1460	n/a	65535	65535	no	no
20081016	54.147.61.79	1460	n/a	65535	65535	no	no
20090115	126.14.171.216	1452	n/a	64240	65340	no	no
20090331	215.35.134.36	n/a	n/a	n/a	65535	no	no
20090618	58.104.167.176	1460	n/a	65535	65535	no	no
20090820	45.36.231.185	1414	n/a	65535	65535	no	no

Table 3.16 reports the Windows reneging instances detected. 75 reneging instances were observed in 53 flows from 9 different hosts. This behavior indicates

that multiple TCP flows renege per Windows data receiver. The average number of reneging instances per flow was 1.41 (75/53) which suggests that Windows reneging occurs multiple times per flow. The average bytes reneged per reneging instance was 1371 bytes (~ 1 MSS PDU).

Table 3.16: Windows reneging instances

Date	Anonymized IP	Reneged Flows	Total Reneging Instances	Total Reneged Bytes	Avg. Reneged Bytes
20080430	59.190.212.36	1	1	98	98
20080430	247.9.212.28	1	3	8760	2920
20080515	7.30.83.155	6	20	15085	754
20080619	238.20.116.194	1	1	4096	4096
20081016	54.147.61.79	1	1	1460	1460
20090115	126.14.171.216	1	1	287	287
20090331	215.35.134.36	1	2	3929	1965
20090618	58.104.167.176	1	2	7100	3550
20090820	45.36.231.185	40	44	61975	1409
TOTAL		53	75	102790	1371

To check if reneging Windows data receivers were busy servers or clients, we listed the number of TCP connections analyzed for each reneging data receiver in Table 3.17. For Windows, the reneging data receivers did not seem busy except for the data receiver identified with IP 59.190.212.36. We admit that other TCP flows could be established to the reneging data receivers which were not observed in our traces. Majority of the data receivers (clients) were transferring data from web servers. Two of the data receivers (clients) were transferring data using ephemeral port numbers and one data receiver was a Simple Mail Transfer (SMTP) server.

Table 3.17: Connection numbers for reneging Windows data receivers

Date	Anonymized IP	Flows using SACKs			Flows not using SACKs
		2 minute	2 minute	10 minute	10 minute
20080430	59.190.212.36	6	0	40	882
20080430	247.9.212.28	1	1		
20080515	7.30.83.155	1	0	10	13
20080619	238.20.116.194	1			
20081016	54.147.61.79	0	1		
20090115	126.14.171.216	1	0	1	16
20090331	215.35.134.36	1		1	2
20090618	58.104.167.176	2			
20090820	45.36.231.185	9		73	56

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

For the Windows reneging instances, two types of reneging behaviors were observed. The first type is more common and observed in 49 reneging flows. In the first type of reneging, only a single out-of-order segment was reneged and the consecutive out-of-order data were not SACKed even though these data are known to be in the receive buffer. This type of reneging is detailed with an example reneging instance shown in Figure 3.13. The second type of reneging is observed in 4 flows. This type of reneging behavior is similar to FreeBSD reneging behavior shown in Figure 3.12.

Figure 3.13 shows a Windows reneging instance that occurred on 2008/05/15 which is an example of the first type of Windows reneging behavior. The initial state of the data receiver's receive buffer is known as stateACK 74511. First, data packets

(#1, 74511-75871) through (#7, 81321-81708) are monitored at the intermediate router. The ack (#9, ACK 75871 SACK 77231-78591) informs the data sender that data packet (#1) is received in order and data packet (#3, 77231-78591) is received out-of-order. The state is updated to stateACK 75871, stateSACK 77231-78591. The next ack (#10, ACK 77231) in acknowledges the receipt of data packet (#2, 75871-77231) in order. Unfortunately, ACK is increased to the left edge of previously SACKed out-of-order data (stateSACK 77231-78591) giving the impression that data are reneged. RenegDetect v2 marks 77231-78591 as MISSING. Next, 4 duplicate acks are observed (#11, #12, #13, and #14 ACK 77231). We believe these duplicate acks are sent when data packets (#4, #5, #6, and #7) are each received out-of-order at the data receiver. When the data sender retransmits the MISSING data, data packet (#15, 77231-78591), ACK is increased beyond the MISSING data. Therefore, we conclude that reneging occurred. Interestingly, ACK is increased to 81708 after the retransmission which confirms that data packets (#4, #5, #6, and #7) are received out-of-order. Even though data packets (#4, #5, #6, and #7) were received out-of-order, the data receiver misbehaved and did not report out-of-order received data with SACKs.

3.2.5.2 SIGCOMM 2008 Traces

In this section, we analyze the reneging in SIGCOMM 2008 conference traces collected at August, 2008. First, we describe the topology and how the traces are collected in Section 3.2.5.2.1. Later, in Section 3.2.5.2.2, we present our findings.

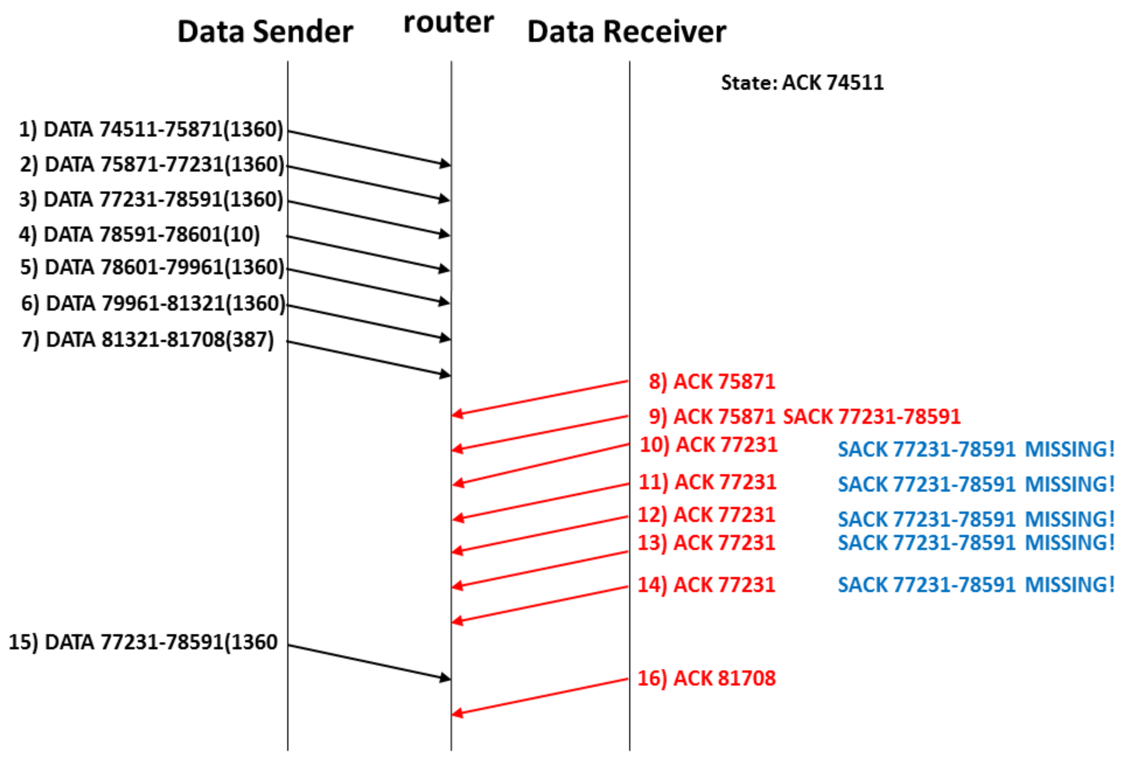

Data Sender
router
Data Receiver
State: ACK 74511
1) DATA 74511-75871(1360)
2) DATA 75871-77231(1360)
3) DATA 77231-78591(1360)
4) DATA 78591-78601(10)
5) DATA 78601-79961(1360)
6) DATA 79961-81321(1360)
7) DATA 81321-81708(387)
8) ACK 75871
9) ACK 75871 SACK 77231-78591
10) ACK 77231
SACK 77231-78591 MISSING!
11) ACK 77231
SACK 77231-78591 MISSING!
12) ACK 77231
SACK 77231-78591 MISSING!
13) ACK 77231
SACK 77231-78591 MISSING!
14) ACK 77231
SACK 77231-78591 MISSING!
15) DATA 77231-78591(1360
16) ACK 81708

Figure 3.13: A Windows reneging instance

3.2.5.2.1 Description of Traces

The SIGCOMM 2008 traces consist of three types of traces: (a) wireless (802.11a): collected from eight 802.11a monitors placed at the four corners of the main conference hall, (b) Ethernet: the packets captured between the Network Address Translator (NAT) and the Access Point (AP), and (c) Syslog from Access Point. For our reneging investigation, we were interested in (a) wireless traces, and (b) Ethernet traces, because only these traces contained TCP traffic using SACKs. In traces, all IP addresses of were anonymized using the tcpmkpub tool; and DHCP assigned IPs for local hosts in the 26.12.0.0/16 and 26.2.0.0/16 subnets after the anonymization process. For more information on the traces, see [Sigcomm 2008].

Wireless traces were collected for four days starting from 08/18/2008 to 08/21/2008 on eight 802.11a monitors where some TCP flows were captured on multiple 802.11a monitors. Ethernet traces are more complete, and were collected for five days between 08/17/2008 and 08/21/2008.

Table 3.18 and 3.19 present the statistics for the percentage of data for the TCP flows using SACKs in wireless and Ethernet traces, respectively. Table 3.18 indicates that the percentages of data that falls between (0, 95) interval (columns 4 and 5) ranges from 28.19% to 48.24% in the wireless traces. This behavior implies that gaps in the data were observed due to packet losses during trace capture. We ignored these wireless traces since missing data/retransmissions would bias the results in favor of not reneging instances.

Table 3.18: Percentage of data monitored in wireless traces

Date	100%	[95, 100)%	(0, 95)%	0%	Multiple flows
20080818	36.44	15.31	43.23	2.41	1.97
20080819	52.06	4.88	29.84	5.55	7.03
20080820	60.13	4.16	22.21	5.98	5.85
20080821	37.90	2.35	36.49	11.75	7.05

Table 3.19: Percentage of data monitored in Ethernet traces

Date	100%	[95, 100)%	(0, 95)%	0%	Multiple flows
20080817	88.24	0.00	0.00	0.00	11.76
20080818	93.38	1.07	0.32	0.64	4.59
20080819	92.82	0.13	0.06	0.31	6.68
20080820	93.08	0.19	0.02	0.16	6.55
20080821	94.01	0.09	0.00	0.14	5.76

In the Ethernet traces shown in Table 3.19, the data that falls between (0, 95) interval (columns 4 and 5) ranges from 0.00% to 0.96%. Recall that the Ethernet data collection monitor was placed between the AP and the NAT, and therefore included all the TCP flows from the wireless traces. Since the wireless traces contain gaps in the data, we only analyzed Ethernet traces for reneging analysis.

3.2.5.2.2 Results

In this section, we document the frequency of reneging in the SIGCOMM 2008 Ethernet traces. Table 3.20 presents the number of TCP flows using SACKs and the frequency of reneging in the Ethernet traces. The columns of the table show the date (in yyyymmdd format), the number of TCP flows using SACK blocks analyzed, the number of candidate reneging flows, and the number of reneged flows, respectively. The candidate reneging flows are those that satisfy the following two conditions: (a) some SACK block(s) were MISSING and (b) data retransmissions for the MISSING SACK block(s) were observed. Each candidate reneging flow was analyzed by hand in detail using wireshark [Wireshark] to determine if reneging happened or if the candidate reneging instance was an instance of a misbehavior. Upon analysis, we found that all of the candidate reneging instances were misbehavior instances. Out of 42 candidate reneging flows (0.27% of all flows) analyzed, no flows reneged.

Table 3.20: Reneging frequency of SIGCOMM 2008 traces

Date	Flows using SACKs	Candidate Reneging Flows	Reneged Flows
20080817	45	0	0
20080818	1791	34	0
20080819	2974	2	0
20080820	8858	4	0
20080821	2015	2	0
TOTAL	15683	42 (0.27%)	0 (0.00%)

In our analysis on SIGCOMM 2008 traces, we found the frequency of reneging to be 0%. This result suggests that reneging is a rare event.

3.2.5.3 Lawrence Berkeley National Laboratory (LBNL) Traces

In this section, we present the reneging frequency of LBNL enterprise traces captured between October, 2004 and January, 2005. First, we describe the traces in Section 3.2.5.3.1. Later, in Section 3.2.5.3.2, we present the results.

3.2.5.3.1 Description of Traces

LBNL traces characterize internal enterprise traffic recorded at a medium-sized site. The traces (11GB) span more than 100 hours of activity from a total of several thousand internal hosts where the IP addresses of the internal hosts were anonymized using tcpmkpub tool. For more information on the traces, see [LBNL 2004].

The enterprise traces were collected for 5 days from October, 2004 to January, 2005. Table 3.21 presents the statistics for the percentage of data for the TCP flows using SACKs in the enterprise traces. The data that falls between (0, 95) interval (columns 4 and 5) ranged from 0.08% to 2.09%. Those flows were ignored along with the traces containing multiple TCP flows (column 6) for reneging analysis.

Table 3.21: Percentage of data monitored in LBNL traces

Date	100%	[95, 100)%	(0, 95)%	0%	Multiple flows
20041004	96.03	0.90	0.18	1.91	0.98
20041215	97.26	0.00	0.01	0.16	2.57
20041216	95.95	0.05	0.02	0.07	3.92
20050106	97.05	0.20	0.00	0.08	2.67
20050107	96.36	0.09	0.02	0.13	3.39

3.2.5.3.2 Results

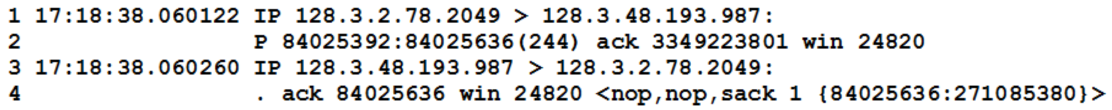
In this section, we present the results of reneging analysis of the enterprise traces provided by LBNL. Table 3.22 presents the frequency of reneging in the LBNL traces. Out of 16 candidate reneging flows (0.06% of all flows), no flows reneged. We report that all the candidate reneging flows were instances of SACK generation misbehaviors.

Table 3.22: Reneging frequency of LBNL traces

Date	Flows using SACKs	Candidate Reneging Flows	Reneged Flows
20041004	2684	1	0
20041215	8134	1	0
20041216	5757	2	0
20050106	4822	6	0
20050107	4357	6	0
TOTAL	25754	16 (0.06%)	0 (0.00%)

In [Blanton 2008], the author also analyzed LBNL traces to report the frequency of reneging. Reneging instances were detected when an ACK increased in the middle of a previously reported SACK. Out of 26,589 TCP flows analyzed, the author reported no instances of reneging. The results of both analyses (our and [Blanton 2008]) are the same: the frequency of reneging reported in LBNL traces is 0.00%. For the same traces, we analyzed less number of TCP flows (25754) since the traces having gaps in the data or containing multiple flows were discarded.

In [Blanton 2008], the author defined a flow as “bogus” if a SACK information was significantly outside of the analyzed sequence space. The author reported 3 “bogus” reneging flows in the LBNL traces. Figure 3.14 shows an example tcpdump output for such a “bogus” reneging flow. The ack (lines 3, 4) notifies the data sender that the data segment (lines 1, 2) was received in order. Unfortunately, the left edge of the reported SACK block 84025636-271085380 is same as the ACK. This behavior was detected as a reneging instance in [Blanton 2008]. Note that the SACK block in Figure 3.14 claimed that 187,059,744 (!) bytes were in the receive buffer. We believe that this behavior is another instance of a misbehaving TCP stack. RenegDetect v2 also detected those “bogus” reneging flows in the LBNL traces. Unlike [Blanton 2008], RenegDetect v2 identified 4 flows as “bogus” and did not report these “bogus” flows as candidate reneging flows since no data retransmissions were observed for the MISSING data.



```
1 17:18:38.060122 IP 128.3.2.78.2049 > 128.3.48.193.987:
2                 P 84025392:84025636(244) ack 3349223801 win 24820
3 17:18:38.060260 IP 128.3.48.193.987 > 128.3.2.78.2049:
4                 . ack 84025636 win 24820 <nop,nop,sack 1 {84025636:271085380}>
```

Figure 3.14: An example “bogus” reneging instance

In our analysis on LBNL traces, we found the frequency of reneging to be 0%. This result suggests that reneging is a rare event.

3.3 Conclusion

To document the frequency of TCP reneging in trace data, we proposed a mechanism to detect reneging instances. The proposed mechanism is based on how an SCTP data sender infers reneging. A state of the receive buffer is constructed at an intermediate router and updated through new acks. When an inconsistency occurs between the state of the receive buffer and a new ack, reneging is detected. We implemented the proposed mechanism as a tool called RenegDetect.

While verifying RenegDetect with real TCP flows, we discovered that some TCP implementations were generating SACKs incompletely under some circumstances giving a false impression that reneging was happening. To identify reneging instances more accurately, we updated RenegDetect to better analyze the flow of data, in particular, to analyze data retransmissions which are a more definitive indication that reneging happened.

Our initial hypothesis was that reneging rarely if ever occurs in practice. For that purpose, TCP traces from three domains (Internet backbone (CAIDA), wireless (SIGCOMM), enterprise (LBNL)) were analyzed using RenegDetect.

Contrary to our initial expectation that reneging is extremely rare event, trace analysis demonstrated that reneging does happen. Therefore, we could not reject our initial hypothesis H_0 that P(reneging) $< 10^{-5}$. Since reneging instances were found, analyzing 300K TCP flows were no longer necessary. As a result, we ended up analyzing 202,877 TCP flows using SACKs from the three domains. Table 3.23 reports the frequency of TCP reneging in the three domains. In the TCP flows using SACKs, we detected 104 reneging flows. Based on these observations, we estimated with 95% confidence that the true average rate of reneging is in the interval [0.041%, 0.059%], roughly 1 flow in 2,000 (0.05%).

Table 3.23: Frequency of reneging

Trace	Flows using SACKs	Linux Reneging	FreeBSD Reneging	Windows Reneging	Total Reneging
CAIDA	161440	40	11	53	104
SIGCOMM	15683	0	0	0	0
LBNL	25754	0	0	0	0
TOTAL	202877	40 (0.02%)	11 (0.00%)	53 (0.03%)	104 (0.05%)

The frequency of TCP reneging we found, 0.05%, is greater than the results in [Blanton 2008] where the frequency of reneging is reported as 0.017%. Together the results of these two studies allow us to conclude that reneging is a rare event.

In the 104 reneging flows, a total of 200 reneging instances were detected. This behavior suggests that multiple reneging instances occur per reneging flow. For each reneging flow, we tried to fingerprint the operating system of the reneging data receiver, and generalize reneging behavior of Linux, FreeBSD, and Windows data receivers.

In this study, we investigated the frequency of TCP reneging to conclude if TCP's design to tolerate reneging is correct. If we could document that reneging never occurs, TCP had no need to tolerate reneging. However, reneging occurs rarely (less than 1 flow per 1000), we believe the current handling of reneging in TCP can be improved.

TCP is designed to tolerate reneging by defining a retransmission policy for a data sender [RFC2018] and keeping the SACKed data in the data sender's send buffer until cumulatively ACKed. With this design, if reneging does not happen or happens rarely, SACKed data are unnecessarily stored in the send buffer wasting operating system resources.

To understand the potential gains for a protocol that does not tolerate reneging, SCTP's NR-SACKs (Non-Renegable SACKs) are detailed in Section 1.2.2. With NR-SACKs, an SCTP data receiver takes the responsibility for non-renegable data (NR-SACKed), and, an SCTP data sender needs not to retain copies of NR-SACKed data in its send buffer until cumulatively ACKed. Therefore, memory allocated for the send buffer is better utilized with NR-SACKs. NR-SACKs also improve end-to-end application throughput. When the send buffer is full, no new data can be transmitted even when congestion and flow control mechanisms allow. When NR-SACKed data are removed from the send buffer, new application data can be read and potentially transmitted.

If current TCP was designed not to tolerate reneging, the send buffer utilization would be always optimal, and the application throughput might be improved for data transfers with constrained send buffers (assuming asymmetric buffer sizes (send buffer < receive buffer) and no auto-tuning). Unfortunately, TCP is designed to tolerate reneging.

Let us compare TCP's current design to tolerate reneging with a TCP that does not support reneging using the results from our reneging analysis. With current design, TCP tolerates reneging to achieve the reliable data transfer of 104 reneging flows. The 202,773 non-reneging flows waste main memory allocated to send buffer and potentially achieve lower throughput.

I argue that the current design to tolerate reneging is wrong since reneging is a rare event. Instead, I suggest that the current semantics of SACKs should be changed from advisory to permanent prohibiting a data receiver to renege. If a data receiver does have to take back memory that has been allocated to received out-of-order data, I 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 benefit from better send buffer utilization and possible increased throughput. The increased throughput is only possible for data transfers with constrained send buffers (assuming asymmetric buffer sizes (send buffer < receive buffer) and no auto-tuning) and needs modifications in TCP's send buffer management.

Initially, reneging was thought as a utility mechanism to help an operating system to reclaim main memory back under low-memory situations. In our investigation, we found that the average main memory returned to the reneging operating system per reneging instance is on the order of 2 TCP segments (2715, 3717, and 1371 bytes for Linux, FreeBSD, and Windows operating systems, respectively.) This average amount of main memory reclaimed back to operating system seems relatively insignificant. For example, to reclaim 3MB of main memory back to FreeBSD, 846 simultaneous TCP flows each having 3717 bytes of out-of-order data would need to be reneged. On the other hand, our experimentation with FreeBSD showed that terminating a single TCP flow established to Apache web server releases ~3MB of main memory in FreeBSD. Therefore, I believe that RESETing a TCP flow is a better strategy to help an operating system rather than the current handling of reneging.

I also had a chance to discuss why reneging is tolerated in TCP with Matt Mathis, the main editor of [RFC2018]. He told me that the semantics of SACKs are advisory since a reliable data transfer would fail if SACKs were permanent and some TCP stacks implement SACKs incorrectly. By specifying SACKs advisory, TCP is more robust to SACK implementations having bugs. I argue that this design choice is wrong. Similarly, a TCP stack implementing a wrong ACK mechanism would cause a data transfer to fail. I believe it is the protocol implementor's responsibility to provide a conformant implementation. In my opinion, the protocols should be specified to achieve the best performance and not be designed to tolerate incorrect implementations. I argue that TCP's current mechanism to tolerate reneging achieves a lower memory utilization when compared to a TCP with no reneging support and should be changed.

Chapter 4

RENEGING SUPPORT IN OPERATING SYSTEMS

This chapter presents reneging support within the following operating systems: FreeBSD, Linux (Android), Mac OS X, OpenBSD, Solaris and Windows. Reneging has been studied only once in the research community to report its frequency [Blanton 2008], but the causes of reneging are unknown. The general assumption is that reneging happens when an operating system goes low on memory to help the operating system recover and resume normal operation. But no one knows if this assumption is true. Our objective is to document the circumstances of reneging in detail for operating systems with reneging support. For that, various TCP stacks are inspected and the interactions between the TCP stack and operating system during reneging are reported. Once the circumstances of reneging are better understood, a tool to cause a remote host to renege can be implemented. In Chapter 5, such a tool is presented, and three operating systems are purposefully reneged to inspect the consequences on the operating system and its transport layer connections. In this chapter, we first investigate what causes reneging.

To determine which operating systems to study for reneging, we decided to inspect those operating systems which are both popular and support SACKs [RFC2018]. Table 4.1 presents the operating systems with at least 0.01% market share on 10/21/2009 reported by www.netmarketshare.com [Market]. Microsoft's Windows occupies the major portion of the market share (greater than 91%). Unfortunately, Microsoft's operating systems are not open source, so their TCP stack cannot be inspected for reneging. To learn more about reneging support in Windows, I contacted implementors of Microsoft's TCP stack and asked if the stack has any support for reneging. Their responses are discussed in Section 4.1.

Table 4.1: Market share of popular operating systems in 2009 [Market]

Operating System	Market Share	RFC 2018 Support	Reneging
Windows XP	71.51%	yes	
Windows Vista	18.62%	yes	yes
Mac OS X 10.5	3.03%	yes	yes
Mac OS X 10.4	0.96%	yes	yes
Windows 2000	0.85%	yes	
Linux	0.95%	yes	yes
iPhone	0.35%		
Mac OS X Mach-O	0.08%	yes	yes
Windows 98	0.11%	yes	
Windows ME	0.06%	yes	
iPod	0.07%		
Windows NT	0.10%	no	
Java ME	0.30%		
Android 1.5	0.02%	yes	yes
Symbian	0.15%		
Windows CE	0.04%		
PLAYSTATION 3	0.02%		
PSP	0.01%		
BlackBerry	0.02%		
FreeBSD	0.01%	yes	yes
Total	97.26%		

The second most popular operating system after Microsoft's Windows is Apple's Mac OS X. Reneging support in Mac OS X is detailed in Section 4.2.

Linux is the third most popular operating system. Reneging support for Linux and Linux-based Android is presented in Section 4.3.

Reneging is expected to happen on hosts which go low on main memory. Therefore, a web server with thousands of active TCP connections is a stronger candidate to renege rather than a web client averaging a few active TCP connections at the time. So in addition to operating systems with a significant market share, we decided to inspect operating systems that are used by busy web servers.

One can argue that most web servers are data senders, and reneging is expected to take place at a data receiver. So why should we inspect TCP stacks of the operating systems hosting web servers? Web pages such as www.rapidshare.com, www.flickr.com, photobucket.com, imageshack.us, www.dailymotion.com and megaupload.com provide online data storage services to their users. These web pages play the role of both a data sender and a data receiver. Hence, reneging support should be investigated for those operating systems.

To find out which operating systems are used by popular web servers, I ran the Network Mapper (nmap) tool [Nmap] on [Alexa]'s most visited 100 web pages in 2009. Nmap can detect the operating system and services of a remote host. Table 4.2 presents the results for operating systems and services detected by nmap for the most visited 100 web pages in 2009. For the Microsoft web pages, www.msn.com and www.microsoft.com, nmap could not detect the operating system running. We can infer that those web pages are hosted on a Windows operating system by simply inspecting the services used (Microsoft IIS Webserver.)

Several top 100 web pages are hosted on FreeBSD, Linux, OpenBSD, and Windows. Reneging support for FreeBSD and OpenBSD is detailed in Sections 4.4 and 4.5, respectively.

Solaris is an operating system with SACK support [RFC2018] that we tested in Chapter 2 for proper TCP SACK generation. While analyzing the TCP stack of

Solaris, a cause for reneging was found accidentally. Section 4.6 details reneging in Solaris.

Table 4.2: Nmap detected operating systems of some of the Alexa's Top Sites

Rank	Domain	Operating System	Services
1	google.com	OpenBSD 4.0	Google Httpd 2.0 (GFE)
3	yahoo.com	FreeBSD 6.3	
5	live.com	Linux 2.6.5-2.6.12	Akamai SSH Server-VII
6	wikipedia.com	Linux 2.6.9-2.6.27	Apache httpd
8	msn.com	UNKNOWN	Microsoft IIS webserver 6.0
16	microsoft.com	UNKNOWN	Microsoft IIS webserver 7.5
21	rapidshare.com	Linux 2.6.15 - 2.6.26	http?
25	amazon.com	OpenBSD 4.0	http?
32	flickr.com	Linux 2.4.32	Apache httpd
34	craigslist.org	FreeBSD 6.2	http?
42	photobucket.com	Linux 2.4.31 - 2.4.34	Apache httpd
62	imageshack.us	Linux 2.6.9 - 2.6.24	lighttpd 1.5.0
83	dailymotion.com	Linux 2.6.9	http?
85	megaupload.com	Linux 2.6.15 - 2.6.26	Apache httpd

In the following sections, variables, functions, structures and file names related to the TCP implementations are shown in italics.

4.1 Reneging in Microsoft's Windows

Microsoft's operating system code is not publicly available. To gain insight into reneging behavior in Windows systems, I contacted Dave MacDonald, the author of Microsoft Windows 2000 TCP/IP Implementation Details [MacDonald 2000]. Dave confirmed that reneging is not possible in Windows 2000, XP and Server 2003.

Vista and its successors (Windows Server 2008 and 7) have a brand new TCP stack. Dave stated that "in Vista+ (Vista and its successors), the only time we renege on reassembly data is if we think the memory consumption of total reassembly data in relation to the global memory limits is significant.” From the emails exchanged, I believe Vista’s reneging mechanism makes sure a connection maintains a minimum amount of forward progress in its end-to-end data transfer. If forward progress happens, reneging does not occur. If no forward progress happens for a maximum time, reneging is invoked and out-of-order data are consequently discarded. Recall that the Microsoft’s TCP stack code is not publicly available, so our conclusion for reneging behavior in Windows is only a conjecture.

Dave stated that the purpose of the reneging mechanism in Windows is to protect the operating system against denial of service (DoS) attacks where attackers force Windows to create state and exhaust resources. Initially, reneging is thought of as a mechanism that helps an operating system which goes low on system resources. Using reneging, an operating system would reclaim some resources back to resume its normal operation. In Windows, on the other hand, reneging seems to have a different purpose: to protect the operating system from DoS attacks similar to SYN flood. In a SYN flood attack, attackers send SYN requests to a victim to open fake connections and consume the victim’s resources, and in the extreme make the victim’s services unavailable. To consume even more resources, an attacker could fill open TCP connections with out-of-order data thus using up memory for receive buffers. To protect from such attacks, Windows uses reneging as an attack protection mechanism.

In Section 5.4, Windows Vista and 7 hosts are reneged and the consequences of reneging are presented.

4.2 Reneging in Mac OS X

This section details reneging support in Mac OS operating system. The Mac OS X kernel is called X is Not Unix (XNU) [Singh 2003]. The TCP/IP stack of XNU is based on XNU's Berkeley Software Distribution (BSD) component whose primary reference codebase is FreeBSD (5.x). XNU has the same mechanism for turning on/off data reneging as does FreeBSD which is detailed in Section 4.4. A major difference between Mac OS and FreeBSD is that currently reneging is off by default in Mac OS X (xnu 1699.24.8) [MacOS] while on by default in FreeBSD. The code segment in Figure 4.1 sets the default behavior for reneging in Mac OS X to off (lines 2, 3), defined in *bsd/netinet/tcp_subr.c*.

```
1 static int        do_tcpdrain = 0;
2 SYSCTL_INT(_net_inet_tcp, OID_AUTO, do_tcpdrain, CTLFLAG_RW | CTLFLAG_LOCKED,
3     &do_tcpdrain, 0,
4     "Enable tcp_drain routine for extra help when low on mbufs");
```

Figure 4.1: Setting up the default reneging behavior in Mac OS X

$$\begin{tikzcd}
\text{mbuf\_slab\_alloc} \arrow[d] \\
\text{mbuf\_sleep} \arrow[d] \\
\text{m\_reclaim} \arrow[d] \\
\text{pfslowtimo} \arrow[d] \\
\text{tcp\_drain} \arrow[d] \\
\text{m\_freem}
\end{tikzcd}$$

Figure 4.2: Call graph for reneging in Mac OS X

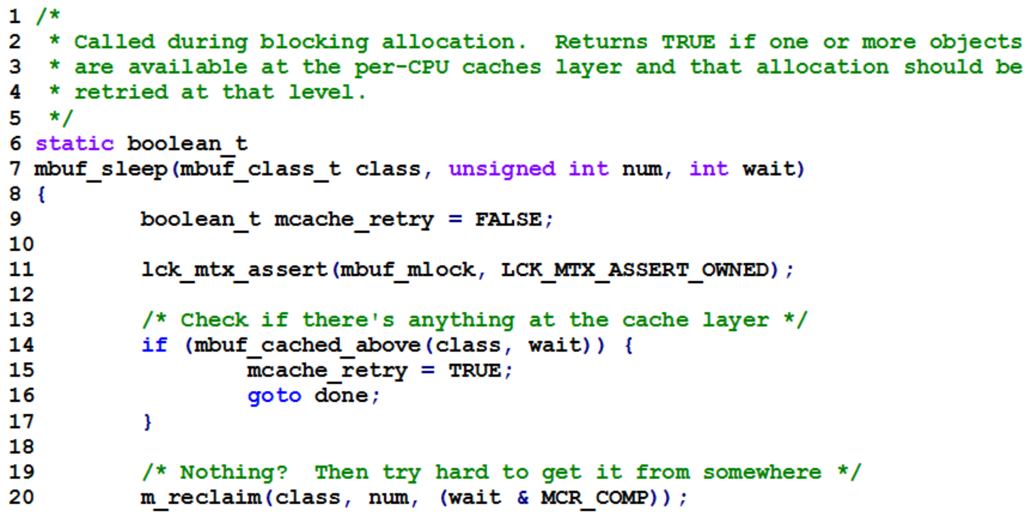
The call graph in Figure 4.2 summarizes the function calls causing a TCP data receiver to renege in Mac OS X.

```
(void) freelist_populate(class, 1,
    (wait & MCR_NOSLEEP) ? M_DONTWAIT : M_WAIT);

if (m_infree(class) > 0)
        continue;

/* Check if there's anything at the cache layer */
if (mbuf_cached_above(class, wait))
        break;

/* watchdog checkpoint */
mbuf_watchdog();

/* We have nothing and cannot block; give up */
if (wait & MCR_NOSLEEP) {
        if (!(wait & MCR_TRYHARD)) {
                m_fail_cnt(class)++;
                mbstat.m_drops++;
                break;
        }
}

/*
 * If the freelist is still empty and the caller is
 * willing to be blocked, sleep on the wait channel
 * until an element is available.  Otherwise, if
 * MCR_TRYHARD is set, do our best to satisfy the
 * request without having to go to sleep.
 */
if (mbuf_worker_ready &&
    mbuf_sleep(class, need, wait))
        break;
```

Figure 4.3: mbuf_slab_alloc() function in Mac OS X

In Mac OS X, caches of mbufs and mbuf clusters exist per CPU. Mbufs/mbuf clusters are structures which store network packets such as Ethernet, IP, and TCP PDUs. Both mbufs and mbuf clusters are defined by rudimentary object type in Mac OS X. Allocation requests for a rudimentary object are first satisfied from the CPU cache using *mcache_alloc()* or *mcache_alloc_ext()* functions. When not enough mbufs or mbuf clusters exist in the CPU cache, *mbuf_slab_alloc()*, defined in *bsd/kern/uipc_mbuf.c*, is used to allocate rudimentary objects from a global freelist of the slab layer. Figure 4.3 shows a code segment from the *mbuf_slab_alloc()* function. If the global freelist is empty, *mbuf_slab_alloc()* attempts to populate it (line 1) first. If the attempt to populate the freelist fails, the *mbuf_sleep()* function is called for the blocking allocation calls (line 31).

mbuf_sleep(), defined in *bsd/kern/uipc_mbuf.c* and shown in Figure 4.4, is called during a blocking allocation. *mbuf_sleep()* tries to serve the request from the CPU cache layer first (line 14). If the request cannot be allocated from the cache layer, *m_reclaim()* is invoked (line 20).



```
1 /*
2  * Called during blocking allocation.  Returns TRUE if one or more objects
3  * are available at the per-CPU caches layer and that allocation should be
4  * retried at that level.
5  */
6 static boolean_t
7 mbuf_sleep(mbuf_class_t class, unsigned int num, int wait)
8 {
9         boolean_t mcache_retry = FALSE;
10
11        lck_mtx_assert(mbuf_mlock, LCK_MTX_ASSERT_OWNED);
12
13        /* Check if there's anything at the cache layer */
14        if (mbuf_cached_above(class, wait)) {
15                mcache_retry = TRUE;
16                goto done;
17        }
18
19        /* Nothing?  Then try hard to get it from somewhere */
20        m_reclaim(class, num, (wait & MCR_COMP));
```

Figure 4.4:   mbuf_sleep() function in Mac OS X

m_reclaim(), defined in *bsd/kern/uipc_mbuf.c*, sets a global variable called *do_reclaim* to 1 (on) which is used by the *pfslowtimo()* function. *pfslowtimo()*, shown in Figure 4.5 and defined in *bsd/kern/uipc_domain.c*, causes reneging in Mac OS X.

The *pfslowtimo()* is similar to FreeBSD's *mb_reclaim()* function shown in Figure 4.15. The *pr_drain* function of each protocol is invoked (line 21) when *do_reclaim* is set to 1 (on). In Mac OS X, *pr_drain* for TCP is initialized in *bsd/netinet/in_proto.c* with the value *tcp_drain*.

```
1 void
2 pfslowtimo(__unused void *arg)
3 {
4         register struct domain *dp;
5         register struct protosw *pr;
6
7         /*
8          * Update coarse-grained networking timestamp (in sec.); the idea
9          * is to piggy-back on the periodic slow timeout callout to update
10         * the counter returnable via net_uptime().
11         */
12        net_update_uptime();
13
14        lck_mtx_lock(domain_proto_mtx);
15        for (dp = domains; dp; dp = dp->dom_next)
16                for (pr = dp->dom_protosw; pr; pr = pr->pr_next) {
17                        if (pr->pr_slowtimo)
18                                (*pr->pr_slowtimo)();
19                        if ((do_reclaim || (pr->pr_flags & PR_AGGDRAIN)) &&
20                            pr->pr_drain)
21                                (*pr->pr_drain)();
22                }
23        do_reclaim = 0;
24        lck_mtx_unlock(domain_proto_mtx);
25        timeout(pfslowtimo, NULL, hz/PR_SLOWHZ);
26 }
```

Figure 4.5: pfslowtimo() function in Mac OS X

Reneging in Mac OS X happens when the *tcp_drain()* function is called, defined in *bsd/netinet/tcp_subr.c*, by the *pfslowtimo()* function. The *net.inet.tcp.do_tcpdrain* sysctl should be set to on (1) beforehand by a system administrator for data reneging to happen. Remember that reneging is turned off by default in Mac OS X. *tcp_drain()* uses the *m_freem()* function in Figure 4.6 (line 27) to delete the reassembly queue which is formed as an mbuf/mbuf cluster chain. Again the *tcp_drain()* function is similar to its corresponding sibling in FreeBSD shown in Figure 4.18.

```
void
tcp_drain()
{
        if (do_tcpdrain)
        {
                struct inpcb *inpb;
                struct tcpcb *tcpb;
                struct tseg_qent *te;

        /*
         * Walk the tcpbs, if existing, and flush the reassembly queue,
         * if there is one...
         * XXX: The "Net/3" implementation doesn't imply that the TCP
         *      reassembly queue should be flushed, but in a situation
         *      where we're really low on mbufs, this is potentially
         *      usefull.
         */
                if (!lck_rw_try_lock_exclusive(tcbinfo.mtx)) /* do it next time if the
                        return;                                  lock is in use */

                for (inpb = LIST_FIRST(tcbinfo.listhead); inpb;
                        inpb = LIST_NEXT(inpb, inp_list)) {
                                if ((tcpb = intotcpcb(inpb))) {
                                        while ((te = LIST_FIRST(&tcpb->t_segq))
                                                != NULL) {
                                        LIST_REMOVE(te, tqe_q);
                                        m_freem(te->tqe_m);
                                        zfree(tcp_reass_zone, te);
                                        tcp_reass_qsize--;
                                }
                        }
                }
                lck_rw_done(tcbinfo.mtx);

        }
}
```

Figure 4.6: tcp_drain() function in Mac OS X

In Mac OS X, reneging is supported by the operating system, and the function calls that can cause a machine to renege are explained above. By default, reneging is turned off. In conclusion, reneging does not happen in Mac OS X unless enabled by a system administrator beforehand.

4.3 Reneging in Linux

This section explains the reneging support in the Linux and Android operating systems. Android is a Linux-based operating system for mobile devices such as cell phones. The Android kernel is based on Linux kernel 2.6.xx and complies with Linux' reneging behavior explained in this section. The call graph in Figure 4.7 summarizes the function calls which cause reneging.

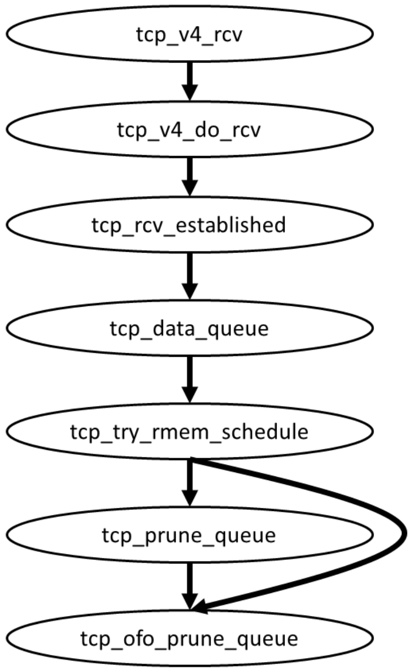


Figure 4.7: Call graph for reneging in Linux

In Linux, out-of-order data are stored in *out_of_order_queue* defined in *include/linux/tcp.h* [Linux]. The *tcp_prune_ofo_queue()* function, shown in Figure 4.8, clears the *out_of_order_queue* of a TCP connection, causing a Linux data receiver to renege (line 12) with the *__skb_queue_purge()* call for the *out_of_order_queue.*

Related SACK information for out-of-order data are deleted using the function *tcp_sack_reset()* (line 20). As a result, reneging in Linux is possible and happens when *tcp_prune_ofo_queue()* is invoked.

```
1 /*
2  * Purge the out-of-order queue.
3  * Return true if queue was pruned.
4  */
5 static int tcp_prune_ofo_queue(struct sock *sk)
6 {
7         struct tcp_sock *tp = tcp_sk(sk);
8         int res = 0;
9
10        if (!skb_queue_empty(&tp->out_of_order_queue)) {
11                NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_OFOPRUNED);
12                __skb_queue_purge(&tp->out_of_order_queue);
13
14                /* Reset SACK state.  A conforming SACK implementation will
15                 * do the same at a timeout based retransmit.  When a connection
16                 * is in a sad state like this, we care only about integrity
17                 * of the connection not performance.
18                 */
19                if (tp->rx_opt.sack_ok)
20                        tcp_sack_reset(&tp->rx_opt);
21                sk_mem_reclaim(sk);
22                res = 1;
23        }
24        return res;
25 }
```

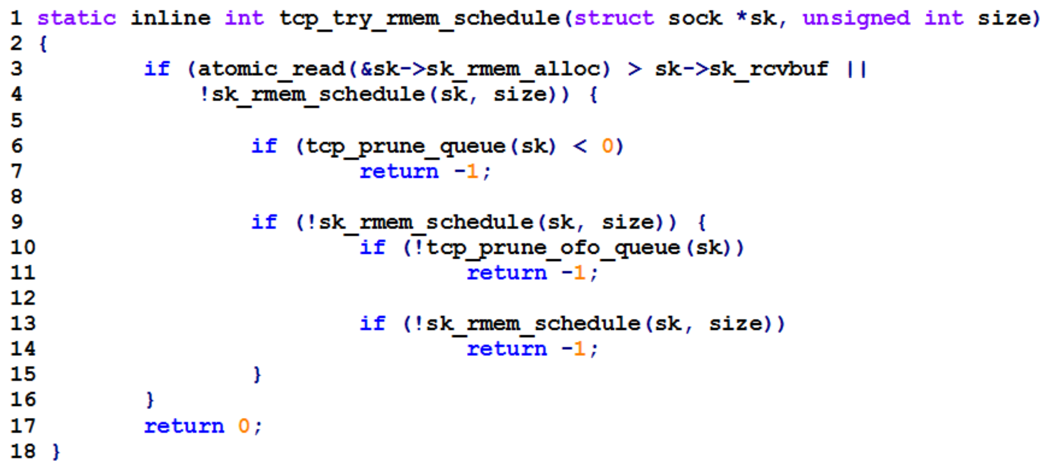
Figure 4.8: tcp_prune_ofo_queue() function in Linux

Linux's TCP stack (specifically kernel version: 2.6) [Linux] includes a function, *tcp_prune_queue()*, that reduces a socket's allocated memory if the socket exceeds its available memory limit. When a socket's allocated memory exceeds the limit, the *tcp_prune_queue()* function can delete out-of-order data from the receive buffer by calling the *tcp_prune_ofo_queue ()* function as shown in Figure 4.9 (line 30).

```
/* Reduce allocated memory if we can, trying to get
 * the socket within its memory limits again.
 *
 * Return less than zero if we should start dropping frames
 * until the socket owning process reads some of the data
 * to stabilize the situation.
 */
static int tcp_prune_queue(struct sock *sk)
{
        struct tcp_sock *tp = tcp_sk(sk);

        if (atomic_read(&sk->sk_rmem_alloc) >= sk->sk_rcvbuf)
                tcp_clamp_window(sk);
        else if (tcp_memory_pressure)
                tp->rcv_ssthresh = min(tp->rcv_ssthresh, 4U * tp->advmss);

        tcp_collapse_ofo_queue(sk);
        if (!skb_queue_empty(&sk->sk_receive_queue))
                tcp_collapse(sk, &sk->sk_receive_queue,
                             skb_peek(&sk->sk_receive_queue),
                             NULL,
                             tp->copied_seq, tp->rcv_nxt);
        sk_mem_reclaim(sk);

        if (atomic_read(&sk->sk_rmem_alloc) <= sk->sk_rcvbuf)
                return 0;

        /* Collapsing did not help, destructive actions follow.
         * This must not ever occur. */

        tcp_prune_ofo_queue(sk);

        if (atomic_read(&sk->sk_rmem_alloc) <= sk->sk_rcvbuf)
                return 0;
```

Figure 4.9:   tcp_prune_queue() function in Linux

Both the *tcp_prune_queue()* and the *tcp_prune_ofo_queue()* functions can be invoked from the *tcp_try_rmem_schedule()* function shown in Figure 4.10. If the *tcp_prune_queue()* call (line 6) returns a negative value (meaning that *tcp_prune_ofo_queue()* is already called by the *tcp_prune_queue()*), *tcp_try_rmem_schedule()* returns. Otherwise, the *tcp_prune_ofo_queue()* can be invoked (line 10). Note that *tcp_try_rmem_schedule()* invokes the *tcp_prune_queue()* and *tcp_prune_ofo_queue()* functions when the memory allocated for receive buffer exceeds the socket's limit for receive buffer (line 3).



```
static inline int tcp_try_rmem_schedule(struct sock *sk, unsigned int size)
{
        if (atomic_read(&sk->sk_rmem_alloc) > sk->sk_rcvbuf ||
            !sk_rmem_schedule(sk, size)) {

                if (tcp_prune_queue(sk) < 0)
                        return -1;

                if (!sk_rmem_schedule(sk, size)) {
                        if (!tcp_prune_ofo_queue(sk))
                                return -1;

                        if (!sk_rmem_schedule(sk, size))
                                return -1;
                }
        }
        return 0;
}
```

Figure 4.10: tcp_try_rmem_schedule() function in Linux

In [Seth 2008], the authors state that calling *tcp_try_rmem_schedule()* eventually may lead to reneging when the socket's memory pool is exhausted and allocation needs to be done from the global TCP memory pool. According to [Seth 2008], this situation happens 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.

4.3.1 Local vs. Global Reneging

In Linux, reneging happens only for the connection(s) exceeding their receive buffer limits. So it is possible to have reneging and non-reneging TCP connections simultaneously. We define this behavior as local reneging. On the other hand, reneging in Mac OS X and FreeBSD happens for all the active connections with out-of-order data. We define this behavior as global reneging. In Chapter 5, we cause operating systems to renege, inspect the consequences of reneging, and in Section 5.5 compare the pros and cons of local vs. global reneging.

In conclusion, reneging happens in Linux and Android when the memory allocated for receive buffer exceeds the memory limit available to the receive buffer.

4.4 Reneging in FreeBSD

This section details reneging support and its implementation in the FreeBSD operating system. FreeBSD comes with built-in reneging support [Freebsd], as does Mac OS X. In FreeBSD, a *sysctl* mechanism enables processes to get and set the kernel state. To turn on/off reneging*,* the *net.inet.tcp.do_tcpdrain* sysctl is used. Reneging can happen when the system runs out of main memory and *net.inet.tcp.do_tcpdrain* is on (1). The code segment shown in Figure 4.11, defined in */usr/src/sys/netinet/tcp_subr.c*, sets the default TCP behavior for reneging to on (1), (lines 2, 3), for FreeBSD.

```
1 static int       do_tcpdrain = 1;
2 SYSCTL_INT(_net_inet_tcp, OID_AUTO, do_tcpdrain, CTLFLAG_RW,
3        &do_tcpdrain, 0,
4        "Enable tcp_drain routine for extra help when low on mbufs");
```

Figure 4.11: Setting up the default reneging behavior in FreeBSD

When *net.inet.tcp.do_tcpdrain* is set to 0 (off), reneging is disabled. In this case, all out-of-order data effectively becomes non-renegable (out-of-order data are never purged from the receive buffer).

The call graph in Figure 4.12 summarizes the function calls/events causing a TCP data receiver to renege in FreeBSD. Now we detail the functions, structures, and event handlers used for reneging.

Two functions in FreeBSD kernel invoke the *vm_lowmem* event when the available main memory goes below a certain threshold. The first function is the *kmem_malloc()* function, defined in */usr/src/sys/vm/vm_kern.c*, and the second function is *vm_pageout_scan()*, defined in */usr/src/sys/vm/vm_pageout.c*. A code segment from *vm_pageout_scan()* is shown in Figure 4.13. When the available main memory goes low, *vm_pageout* daemon (daemon responsible for page replacement) invokes the function *vm_pageout_scan()* to scan main memory to free some pages. If the memory shortage is severe enough, the largest process is killed. The *vm_pageout* daemon uses values that, for the most part, are hard-coded or tunable in order to determine paging thresholds [Bruning 2005]. In such a low memory situation, the *vm_lowmem* event is set (line 19) in Figure 4.13.

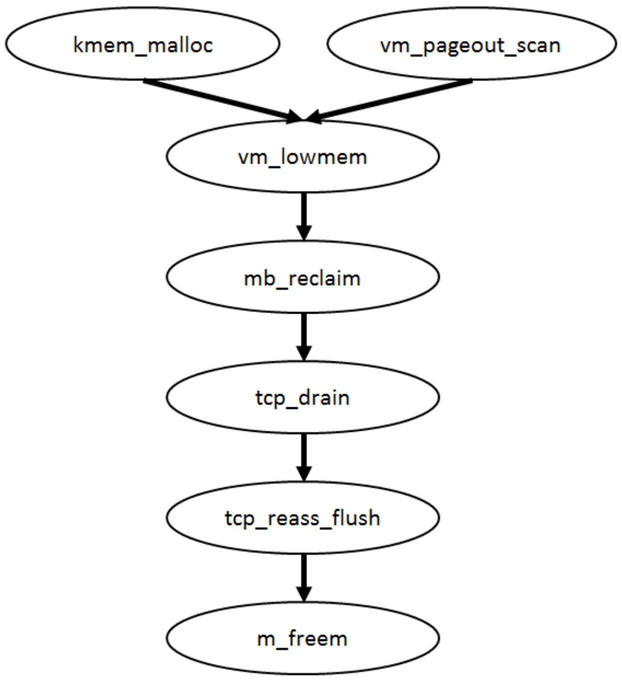


Figure 4.12: Call graph for reneging in FreeBSD

```
/*
 *      vm_pageout_scan does the dirty work for the pageout daemon.
 */
static void
vm_pageout_scan(int pass)
{
        vm_page_t m, next;
        struct vm_page marker;
        int page_shortage, maxscan, pcount;
        int addl_page_shortage, addl_page_shortage_init;
        vm_object_t object;
        int actcount;
        int vnodes_skipped = 0;
        int maxlaunder;

        /*
         * Decrease registered cache sizes.
         */
        EVENTHANDLER_INVOKE(vm_lowmem, 0);
        /*
         * We do this explicitly after the caches have been drained above.
         */
        uma_reclaim();
```

Figure 4.13: vm_pageout_scan() function in FreeBSD

The event handler for *vm_lowmem* is defined and initialized in */usr/src/sys/kern/kern_mbuf.c*. In a low memory situation, first the *vm_lowmem* event is set, and later the *mb_reclaim()* is invoked as a consequence. The registration of the *vm_lowmem* event to the *mb_reclaim()* function is shown (line 6) in Figure 4.14.

```
1 /*
2  * Hook event handler for low-memory situation, used to
3  * drain protocols and push data back to the caches (UMA
4  * later pushes it back to VM).
5  */
6  EVENTHANDLER_REGISTER(vm_lowmem, mb_reclaim, NULL,
7     EVENTHANDLER_PRI_FIRST);
```

Figure 4.14: Event handler for low-memory situation in FreeBSD

The *mb_reclaim()* function, defined in */usr/src/sys/kern/kern_mbuf.c,* is shown in Figure 4.15. The *mb_reclaim()* calls the initialized *pr_drain* function for each protocol (line 20) in each domain.

```
1   /*
2    * This is the protocol drain routine.
3    *
4    * No locks should be held when this is called.  The drain routines have to
5    * presently acquire some locks which raises the possibility of lock order
6    * reversal.
7    */
8   static void
9   mb_reclaim(void *junk)
10   {
11          struct domain *dp;
12          struct protosw *pr;
13
14          WITNESS_WARN(WARN_GIANTOK | WARN_SLEEPOK | WARN_PANIC, NULL,
15              "mb_reclaim()");
16
17          for (dp = domains; dp != NULL; dp = dp->dom_next)
18                  for (pr = dp->dom_protosw; pr < dp->dom_protoswNPROTOSW; pr++)
19                          if (pr->pr_drain != NULL)
20                                  (*pr->pr_drain)();
21  }
```

Figure 4.15: mb_reclaim() function in FreeBSD

/usr/src/sys/sys/protosw.h defines the generic protocol switch table structure that is used for protocol-to-protocol and system-to-protocol communication. This protocol switch table structure, shown in Figure 4.16, is initialized for different protocols supported by FreeBSD such as IP, TCP, UDP and SCTP. The *pr_drain* function pointer for drain routines is defined (line 16).

The protocol switch table structure for TCP is initialized, as shown in Figure 4.17, in */usr/src/sys/netinet/in_proto.c*. Note that the *pr_drain* function pointer is initialized (line 14) with *tcp_drain*. Similarly, the *pr_drain* functions for other protocols such as IP and SCTP are defined with the *ip_drain* and *sctp_drain* functions.

```
struct protosw {
        short   pr_type;                /* socket type used for */
        struct  domain *pr_domain;      /* domain protocol a member of */
        short   pr_protocol;            /* protocol number */
        short   pr_flags;               /* see below */
/* protocol-protocol hooks */
        pr_input_t *pr_input;           /* input to protocol (from below) */
        pr_output_t *pr_output;         /* output to protocol (from above) */
        pr_ctlinput_t *pr_ctlinput;     /* control input (from below) */
        pr_ctloutput_t *pr_ctloutput;   /* control output (from above) */
/* utility hooks */
        pr_init_t *pr_init;
        pr_destroy_t *pr_destroy;
        pr_fasttimo_t *pr_fasttimo;     /* fast timeout (200ms) */
        pr_slowtimo_t *pr_slowtimo;     /* slow timeout (500ms) */
        pr_drain_t *pr_drain;           /* flush any excess space possible */

        struct  pr_usrreqs *pr_usrreqs; /* user-protocol hook */
};
```

Figure 4.16: Protocol switch table structure in FreeBSD

```
{
        .pr_type =              SOCK_STREAM,
        .pr_domain =            &inetdomain,
        .pr_protocol =          IPPROTO_TCP,
        .pr_flags =             PR_CONNREQUIRED|PR_IMPLOPCL|PR_WANTRCVD,
        .pr_input =             tcp_input,
        .pr_ctlinput =          tcp_ctlinput,
        .pr_ctloutput =         tcp_ctloutput,
        .pr_init =              tcp_init,
#ifdef VIMAGE
        .pr_destroy =           tcp_destroy,
#endif
        .pr_slowtimo =          tcp_slowtimo,
        .pr_drain =             tcp_drain,
        .pr_usrreqs =           &tcp_usrreqs
},
```

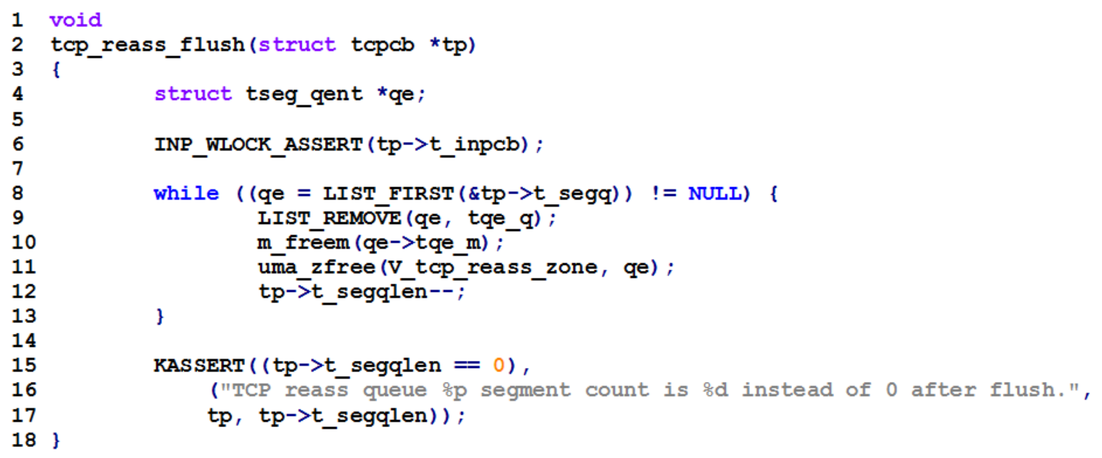
Figure 4.17: Protocol switch table initialization for TCP in FreeBSD

The *tcp_drain()* function causes reneging by deleting all of the reassembly queues of all active TCP connections (global reneging) by calling the *tcp_reass_flush()* function for each queue. The *tcp_drain()*, defined in */usr/src/sys/netinet/tcp_subr.c* and shown in Figure 4.18, goes through all existing TCP connections, and calls *tcp_reass_flush()* for each connection (line 29). Since the reassembly queue of each TCP connection is cleared, the related SACK information (scoreboard) is pruned with a *tcp_clean_sackreport()* call (line 30).

```
1  void
2  tcp_drain(void)
3  {
4          VNET_ITERATOR_DECL(vnet_iter);
5
6          if (!do_tcpdrain)
7                  return;
8
9          VNET_LIST_RLOCK_NOSLEEP();
10         VNET_FOREACH(vnet_iter) {
11                 CURVNET_SET(vnet_iter);
12                 struct inpcb *inpb;
13                 struct tcpcb *tcpb;
14
15         /*
16          * Walk the tcpbs, if existing, and flush the reassembly queue,
17          * if there is one...
18          * XXX: The "Net/3" implementation doesn't imply that the TCP
19          *      reassembly queue should be flushed, but in a situation
20          *      where we're really low on mbufs, this is potentially
21          *      usefull.
22          */
23                 INP_INFO_RLOCK(&V_tcbinfo);
24                 LIST_FOREACH(inpb, V_tcbinfo.ipi_listhead, inp_list) {
25                         if (inpb->inp_flags & INP_TIMEWAIT)
26                                 continue;
27                         INP_WLOCK(inpb);
28                         if ((tcpb = intotcpcb(inpb)) != NULL) {
29                                 tcp_reass_flush(tcpb);
30                                 tcp_clean_sackreport(tcpb);
31                         }
32                         INP_WUNLOCK(inpb);
33                 }
34                 INP_INFO_RUNLOCK(&V_tcbinfo);
35                 CURVNET_RESTORE();
36         }
37         VNET_LIST_RUNLOCK_NOSLEEP();
38 }
```

Figure 4.18: tcp_drain() function in FreeBSD

Shown in Figure 4.19, *tcp_reass_flush()* is defined in */usr/src/sys/netinet/tcp_reass.c*. The function *tcp_reass_flush()* uses *m_freem()* (line 10) to free an entire mbuf chain, including any external storage (mbuf clusters) [FreebsdImpl]. Mbufs and mbuf clusters are structures which store network packets such as Ethernet, IP, and TCP PDUs in FreeBSD. The reassembly queue of a TCP connection is implemented as an mbuf chain of TCP PDUs. A detailed explanation of FreeBSD's network buffers and structures can be found in Section 5.2.1.



```
void
tcp_reass_flush(struct tcpcb *tp)
{
        struct tseg_qent *qe;

        INP_WLOCK_ASSERT(tp->t_inpcb);

        while ((qe = LIST_FIRST(&tp->t_segq)) != NULL) {
                LIST_REMOVE(qe, tqe_q);
                m_freem(qe->tqe_m);
                uma_zfree(V_tcp_reass_zone, qe);
                tp->t_segqlen--;
        }

        KASSERT((tp->t_segqlen == 0),
            ("TCP reass queue %p segment count is %d instead of 0 after flush.",
            tp, tp->t_segqlen));
}
```

Figure 4.19: tcp_reass_flush() function in FreeBSD

In conclusion, FreeBSD is an operating system with built-in support for reneging which can be turned on/off by a system administrator. Reneging would happen when a FreeBSD host goes low on main memory. In Section 5.2, a FreeBSD 8.1 host is reneged and the consequences of reneging are presented.

4.5 Reneging in OpenBSD

OpenBSD is a free, 4.4BSD-based Unix-like operating system with the latest release OpenBSD 5.0 [Openbsd].

As in FreeBSD, */usr/src/sys/sys/protosw.h* defines the generic protocol switch table structure used for protocol-to-protocol and system-to-protocol communication for OpenBSD. The protocol switch table structure for OpenBSD shown in Figure 4.20

is similar to FreeBSD's protocol switch table shown in Figure 4.16 (only minor differences exist between the two protocol switch tables.)

```
struct protosw {
        short   pr_type;                /* socket type used for */
        struct  domain *pr_domain;      /* domain protocol a member of */
        short   pr_protocol;            /* protocol number */
        short   pr_flags;               /* see below */

/* protocol-protocol hooks */
                                        /* input to protocol (from below) */
        void    (*pr_input)(struct mbuf *, ...);
                                        /* output to protocol (from above) */
        int     (*pr_output)(struct mbuf *, ...);
                                        /* control input (from below) */
        void    *(*pr_ctlinput)(int, struct sockaddr *, u_int, void *);
                                        /* control output (from above) */
        int     (*pr_ctloutput)(int, struct socket *, int, int, struct mbuf **);

/* user-protocol hook */
                                        /* user request: see list below */
        int     (*pr_usrreq)(struct socket *, int, struct mbuf *,
                    struct mbuf *, struct mbuf *, struct proc *);

/* utility hooks */
        void    (*pr_init)(void);       /* initialization hook */
        void    (*pr_fasttimo)(void);   /* fast timeout (200ms) */
        void    (*pr_slowtimo)(void);   /* slow timeout (500ms) */
        void    (*pr_drain)(void);      /* flush any excess space possible */
                                        /* sysctl for protocol */
        int     (*pr_sysctl)(int *, u_int, void *, size_t *, void *, size_t);
};
```

Figure 4.20: Protocol switch table structure in OpenBSD

The protocol switch table structure is initialized, as shown in Figure 4.21, in */usr/src/sys/netinet/in_proto.c*. Contrary to FreeBSD, the *pr_drain* function pointer for TCP in OpenBSD is initialized to 0 (NULL) (line 13) instead of a *tcp_drain()* function. This difference reveals that OpenBSD does not have operating system support for reneging (so out-of-order data are non-renegable).

In conclusion, reneging is not possible in OpenBSD.

```
{ SOCK_STREAM,
  &inetdomain,
  IPPROTO_TCP,
  PR_CONNREQUIRED|PR_WANTRCVD|PR_ABRTACPTDIS|PR_SPLICE,
  tcp_input,
  0,
  tcp_ctlinput,
  tcp_ctloutput,
  tcp_usrreq,
  tcp_init,
  0,
  tcp_slowtimo,
  0,
  tcp_sysctl
},
```

Figure 4.21: Protocol switch table initialization for TCP in OpenBSD

4.6 Reneging in Solaris

In Solaris, no built-in support exists for reneging as in FreeBSD or Mac OS X. However, reneging can happen under specific circumstances which are detailed below.

In Solaris, the TCP reassembly queue (queue to store out-of-order data) is referenced by two pointers named *tcp_reass_head* and *tcp_reass_tail*, defined in *common/inet/tcp.h* and shown in Figure 4.22.

```
mblk_t  *tcp_reass_head;        /* Out of order reassembly list head */
mblk_t  *tcp_reass_tail;        /* Out of order reassembly list tail */
```

Figure 4.22: Reassembly queue in Solaris

When an IP packet is received, the data part of the IP PDU is passed to TCP via the *tcp_input_data()* function. The *tcp_input_data()* function passes the new data to the *tcp_reass()* function to either store data in the reassembly queue if the data are out-of-order, or to get all the in-order data when the new data fills the first gap in the reassembly queue. When the in-order data are returned from *tcp_reass()*, they are delivered to the receiving application.

Figure 4.25 shows part of the *tcp_input_data()* function. *tcp_reass()* is called first (line 5). If some out-of-order data still exist in the reassembly queue when *tcp_reass()* returns, a timer called *tcp_reass_timer* (TCP reassembly timer) is restarted with *tcps->tcps_reass_timeout* (timeout value for reassembly timer) value (line 33). The variable *tcps_reass_timeout* is defined in *common/inet/tcp_impl.h* as shown in Figure 4.23. Its default value of *100*SECONDS* is defined in *common/inet/tcp/tcp_tunables.c* as shown in Figure 4.24.

```
#define tcps_reass_timeout              tcps_propinfo_tbl[59].prop_cur_uval
```

Figure 4.23: Definition of tcps_reass_timeout in Solaris

```
{ "_reass_timeout", MOD_PROTO_TCP,
  mod_set_uint32, mod_get_uint32,
  {0, UINT32_MAX, 100*SECONDS}, {100*SECONDS} },
```

Figure 4.24: tcp_propinfo_tbl[59] value in Solaris

When the *tcp_reass_timer* expires after 100 seconds, the *tcp_reass_timer()* function, shown in Figure 4.26 and defined in *common/inet/tcp/tcp_timers.c,* is invoked. All out-of-order data in the reassembly queue are deleted (i.e., reneged) with the *tcp_close_mpp()* call (line 15). Before the out-of-order data are pruned, all related SACK information is cleared with the *tcp_sack_remove()* function (line 12). Since the reassembly queue is emptied, the pointer *tcp_reass_tail* is set to NULL (line 16). In Solaris, reneging happens when the *tcp_reass_timer* timer expires.

```
/*
 * Attempt reassembly and see if we have something
 * ready to go.
 */
mp = tcp_reass(tcp, mp, seg_seq);
/* Always ack out of order packets */
flags |= TH_ACK_NEEDED | TH_PUSH;
if (mp) {
        ASSERT((uintptr_t)(mp->b_wptr - mp->b_rptr) <=
            (uintptr_t)INT_MAX);
        seg_len = mp->b_cont ? msgdsize(mp) :
            (int)(mp->b_wptr - mp->b_rptr);
        seg_seq = tcp->tcp_rnxt;
        /*
         * A gap is filled and the seq num and len
         * of the gap match that of a previously
         * received FIN, put the FIN flag back in.
         */
        if ((tcp->tcp_valid_bits & TCP_OFO_FIN_VALID) &&
            seg_seq + seg_len == tcp->tcp_ofo_fin_seq) {
                flags |= TH_FIN;
                tcp->tcp_valid_bits &=
                    ~TCP_OFO_FIN_VALID;
        }
        if (tcp->tcp_reass_tid != 0) {
                (void) TCP_TIMER_CANCEL(tcp,
                    tcp->tcp_reass_tid);
                /*
                 * Restart the timer if there is still
                 * data in the reassembly queue.
                 */
                if (tcp->tcp_reass_head != NULL) {
                        tcp->tcp_reass_tid = TCP_TIMER(
                            tcp, tcp_reass_timer,
                            tcps->tcps_reass_timeout);
                } else {
                        tcp->tcp_reass_tid = 0;
                }
        }
```

Figure 4.25: tcp_input_data() function in Solaris

```
void
tcp_reass_timer(void *arg)
{
        conn_t *connp = (conn_t *)arg;
        tcp_t *tcp = connp->conn_tcp;

        tcp->tcp_reass_tid = 0;
        if (tcp->tcp_reass_head == NULL)
                return;
        ASSERT(tcp->tcp_reass_tail != NULL);
        if (tcp->tcp_snd_sack_ok && tcp->tcp_num_sack_blk > 0) {
                tcp_sack_remove(tcp->tcp_sack_list,
                    TCP_REASS_END(tcp->tcp_reass_tail), &tcp->tcp_num_sack_blk);
        }
        tcp_close_mpp(&tcp->tcp_reass_head);
        tcp->tcp_reass_tail = NULL;
        TCP_STAT(tcp->tcp_tcps, tcp_reass_timeout);
}
```

Figure 4.26: tcp_reass_timer() function in Solaris

In Solaris, the default value for *tcp_reass_timeout* can be read and modified with the *ndd* (the command to get/set driver configuration parameters). Figure 4.27 shows how to read the default *tcp_reass_timeout* value and change it to 10*SECONDs (10000).

```
nekiz@solaris_chablis:/usr/apache2/2.2/bin# ndd -get /dev/tcp tcp_reass_timeout
100000
nekiz@solaris_chablis:/usr/apache2/2.2/bin# ndd -set /dev/tcp tcp_reass_timeout 10000
```

Figure 4.27: ndd command to change TCP parameters in Solaris

To our best knowledge, a timer for the TCP reassembly queue is not defined in any TCP specification. The best known and most widely used TCP timers are the retransmission timer, the TIME-WAIT timer, the delayed ACK timer, the persist timer, the keep-alive timer, the FIN-WAIT-2 timer, and the SYN-ACK timer. Those timers are enough for TCP to achieve reliable data transfer. None of the open-source operating systems, inspected for reneging in this chapter except Solaris, have a TCP reassembly timer. I inquired what the purpose of this timer was in the Oracle's Developer and Networking forums but nobody replied. I believe the TCP reassembly queue timer in Solaris serves a similar purpose, as does Windows' reneging mechanism: to protect the operating system from a SYN flood-like attack. The reassembly queue is emptied if no data are delivered to receiving application within 100 seconds of receiving any out-of-order data. The allocated resources for the out-of-order data are returned back to the operating system.

To summarize, reneging occurs in Solaris when a data receiver receives some out-of-order data and that data remains in TCP's reassembly queue for at least 100 seconds (the default timeout value). Then, *tcp_reass_timer* timer expires and calls the *tcp_reass_timer()* function which prunes all the data in the reassembly queue. As in Linux, reneging in Solaris is an example of local reneging since only the individual TCP connection(s) are reneged. In Section 5.3, we confirm that reneging happens on a Solaris 11 host when the conditions described above hold.

4.7 Conclusion

In this investigation, several TCP stacks from popular operating systems are inspected to find out the circumstances of reneging. The primary contribution of our investigation is that we found out that operating systems use reneging for different purposes.

Initially, reneging was expected to happen on operating systems that go low on main memory to help the operating system to resume normal operation. FreeBSD supports that type of reneging. In low memory situations, all TCP connections with out-of-order data renege (global reneging) and memory used for the out-of-order data is given back to the operating system.

For Microsoft Windows, reneging is not supported by 2000, XP and Server 2003. Vista+ (Vista, Server 2008, 7) comes with a new TCP stack in which reneging is possible. Reneging in Windows Vista+ is of different type and was introduced to protect a host against DoS attacks. An attacker can open multiple TCP connections and fill each one’s receive buffers with out-of-order data to exhaust system resources to make services unavailable. Reneging happens when the memory consumption of total reassembly data in relation to the global memory limits is significant.

In Mac OS X, reneging is supported by the operating system. Reneging does not happen in Mac OS X unless enabled by a system administrator. As in FreeBSD, Mac OS X employs a global reneging mechanism.

In Linux (Android), reneging happens when the memory allocated for a receive buffer exceeds the memory limit available to the receive buffer. Allocated buffer space for the out-of-order data is freed and returned back to the global TCP memory pool to be used by other TCP connections. Note that only individual connections exceeding the receive buffer limit renege (local reneging).

Reneging is not supported in Solaris but happens to the connections where TCP reassembly queue timer expires (local reneging). To our best knowledge, a timer for the reassembly queue is not defined in the TCP specification. We believe reneging (having a reassembly queue timer) in Solaris has the same purpose as Windows reneging: to protect the operating system against a DoS attack.

Initially, we expected reneging not to be supported by any operating systems. Interestingly, our investigation revealed that five out of six inspected operating systems can renege (FreeBSD, Linux (Android), Apple's Mac OS X, Oracle's Solaris and Microsoft's Windows Vista+.) The only operating system that does not support reneging in our investigation is OpenBSD. We also initially expected that reneging would occur to help operating system to resume normal operation by providing extra memory (FreeBSD). Surprisingly, we discovered that reneging is also used as a protection mechanism against DoS attacks (Solaris, Vista+.) We conclude that reneging is a common mechanism implemented in many of today's popular operating systems.

Chapter 5

CAUSING RENEGING ON A REMOTE HOST

The consequences of reneging on operating systems and active transport connections are unknown. Does reneging help an operating system to resume its operation? Can a reneged TCP connection complete a data transfer? To answer these questions, operating systems and TCP connections should be inspected after reneging. But how can we cause a machine(s) to renege? In the previous chapter, we learn that reneging happens (a) when system resources such as main memory/network buffers become scarce, or (b) when out-of-order data sit in a receive buffer for long time without being delivered to a receiving application. To cause reneging, a tool can exhaust system resources by filling TCP receive buffers of a remote host with out-of-order data and not transmitting in-order data, satisfying (a) and (b), respectively. This chapter presents a tool which causes a remote host to renege, and the tool's application on FreeBSD, Solaris, and Windows operating systems. For those operating systems, the consequences of reneging are detailed by answering the following two questions.

(1) Does reneging help an operating system avoid crashing, thereby resuming normal operation? If yes, we can conclude that reneging is a useful and essential mechanism. On the other hand, after reneging if a machine still cannot resume normal operation (i.e., it crashes), then why bother even implementing reneging?

(2) Can an active TCP connection complete a data transfer successfully when some of the out-of-order data are reneged? When reneging happens, a TCP data receiver deletes all out-of-order data from its receive buffer. In general, a TCP data

sender does not have a mechanism to infer reneging. To tolerate reneging, a sender is expected to discard its SACK scoreboard upon a retransmission timeout, and retransmit bytes at the left edge of the window [RFC2018]. If the TCP sender does not implement tolerating reneging properly, reneging may cause a data transfer to fail.

To answer (1) and (2), operating systems that renege should be analyzed. To analyze a reneging host and its connections, exact timing of reneging needs to be known beforehand. Reneging, in general, is expected to happen under rare circumstances (conditions) when the available main memory/network buffers of a host become scarce. We cannot just sit and wait for reneging to happen. Instead of waiting for a rare event such as reneging to eventually happen, a tool to cause reneging on a remote host can be developed to investigate consequences of reneging in a lab controlled environment. Using such a tool, remote hosts with different operating systems can be analyzed in detail to characterize the consequences of reneging.

Our tool to cause a remote host to renege is called CauseReneg and is detailed in Section 5.1. CauseReneg exhausts a remote host's resources using TCP until the point that reneging is triggered. Figure 5.1 depicts a simple architecture. An attacker runs CauseReneg to attack a remote host (victim). During the attack, the TCP traffic between the attacker and the victim is recorded for later analysis.

The remote host is called a victim since CauseReneg exhausts the victim's operating system resources such as main memory, network buffers, and CPU cycles. CauseReneg is hostile to the victim's operating system and falls into the category of a denial-of-service (DoS) attack tool.

Using CauseReneg, FreeBSD, Solaris and Windows victims are reneged. The consequences of reneging on those operating systems and their TCP connections are presented in Section 5.2, 5.3 and 5.4, respectively. Section 5.5 summarizes our effort to cause reneging on remote hosts.

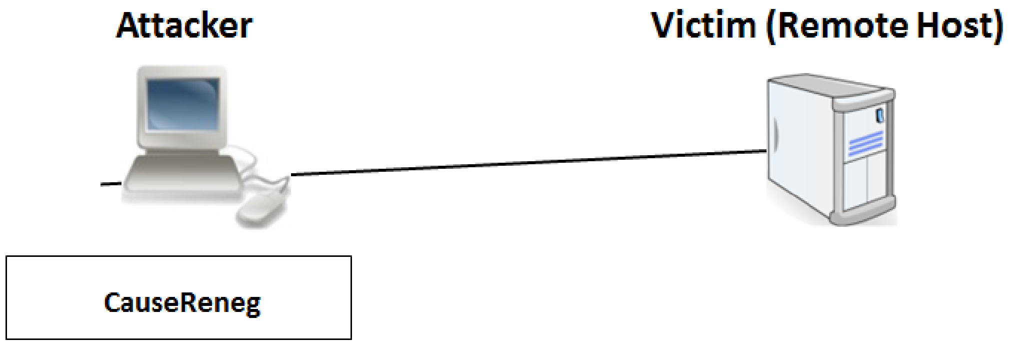


Figure 5.1: Causing a remote host to renege

5.1 A Tool to Cause a Remote Host to Renege: CauseReneg

CauseReneg tries to exhaust a victim’s resources such as main memory and network buffers. Once a victim has reneged, the consequences of reneging to the victim’s operating system and its active transport connections can be documented.

Reneging occurs when a TCP data receiver receives, SACKs, and discards out-of-order data from its receive buffer. To cause a victim to renege, CauseReneg needs to make sure that out-of-order data are present in the receive buffers of the victim. For that, CauseReneg exhausts a victim’s resources by filling a TCP receiver’s receive buffer almost fully with out-of-order data. A victim’s TCP allocates main memory and network buffers to store that out-of-order data in a receive buffer (or a reassembly queue). Since out-of-order data cannot be delivered to the receiving application, resources are held for the time out-of-order data sit in the receive buffer. To exhaust more main memory or network buffers, CauseReneg establishes *n* parallel TCP connections to the victim. As *n* increases, the victim is expected to go low on main memory and network buffers. Eventually, with enough connections, reneging occurs and main memory used for the out-of-order data is reclaimed back to the victim's operating system.

We consider two possible options for CauseReneg to fill a victim's receive buffers with out-of-order data. The options involve either using Dummynet or TCP Behavior Inference Tool (TBIT).

The first option is to use the Dummynet traffic shaper [Dummynet] along the kernel TCP. To create out-of-order data, Dummynet specifies TCP PDUs at the left edge of the window to be dropped for each TCP connection established by CauseReneg to a victim. Since kernel TCP is used, a problem exists with the Dummynet option.

The loss recovery mechanism limits the duration of an attack. TCP's loss recovery mechanism retransmits a dropped TCP PDU *r* times (for example, *TcpMaxDataRetransmissions* in Windows Server 2003 defines *r*=5 by default.) After *r* retransmissions, kernel TCP would terminate a TCP connection. This problem limits the duration of each TCP connection to 1-2 minutes (assuming back to back timeouts, an initial retransmission timeout value (RTO) of 1 second, and *r*=5). When a TCP connection is terminated, the resources allocated to the connection are reclaimed by the victim's operating system. Reneging is expected to happen when the victim's resources are scarce. On the other hand, terminating a connection increases available resources of the victim, and decreases the possibility of reneging. To increase the possibility of reneging, active TCP connections should remain alive to retain resources for longer times.

The second option to implement CauseReneg is to use TBIT [Padhye 2001]. TBIT is a user level traffic generator that produces synthetic TCP PDUs, and does not conform to congestion and loss recovery mechanisms of a standard TCP data sender. TCP PDUs can be sent in any arbitrary order to a TCP receiver. Using TBIT, CauseReneg can avoid sending some bytes at the left edge of the window (say the first 1455 bytes), and fill the rest of the receive buffer with out-of-order data as intended.

The TBIT option does not have the problem of the Dummynet option. As stated above, TBIT does not have to conform to TCP's loss recovery mechanism. So, no retransmissions are needed for the missing TCP PDUs. A victim's TCP receiver has no mechanisms to validate/correct a TCP sender's (TBIT) congestion control or loss recovery mechanisms. A victim's TCP can only dictate flow control which is limited to the advertised receiver window (rwnd). Hence, a victim's TCP would accept any sequence of bytes from CauseReneg when TCP data falls within the rwnd. Once the out-of-order data are received by a victim, a TCP connection is active for at least the keep-alive timer duration. The keep-alive timer is specified to be no less than 2 hours in [RFC1122]. I believe that 2 hours is enough time to cause a victim to renege as compared to Dummynet option's 1-2 minute long TCP connection time.

We consider the TBIT option to be more appropriate for CauseReneg. Thus TBIT is extended with a new test called CauseReneging. CauseReneging fills a victim's TCP receive buffer with out-of-order data. CauseReneg runs *n* CauseReneging TBIT tests to establish *n* TCP connections in parallel with a victim. The number of parallel TCP connections (*n*) used by CauseReneg tool is dynamic and depends on a victim's main memory, available network buffers, and operating system.

In CauseReneg, each CauseReneging test maps to a single TCP connection. The CauseReneging TBIT test is shown in Figure 5.2 and operates as follows:

CauseReneging

1. TBIT establishes a connection to a victim with SACK-Permitted option and Initial Sequence Number (ISN) 10000
2. Victim replies with SACK-Permitted option
3. TBIT sends segment (10001-10006) in order
4. Victim acks the in order data with ACK (10006)
5. TBIT skips sending 1455 bytes (10006-11461) and starts sending *m* consecutive out-of-order segments each 1460 bytes to exhaust main memory
6. Victim acks the out-of-order data with SACKs
7. TBIT sends a 10 byte out-of-order segment after *x* seconds
8. Victim acks the out-of-order data with SACK
9. TBIT sends *m+1* data segments in-order to complete the data transfer
10. Victim acks the in-order data with ACKs/SACKs
11. TBIT sends three RSTs to abort the connection

Now we explain the CauseReneging TBIT test in detail. First, a TCP connection is established to a victim with 3-way handshake (step #1, #2, and #3) with SACK-Permitted option. A 5 byte in-order data is sent to the victim along the ACK (step #3). Next, the victim's receive buffer is filled with *m* out-of-order segments (step #5) based on the advertised window (step #2). As more TCP connections are established to the victim, we expect reneging to happen. Let us assume that reneging happens after *y* seconds. In (step #7), a 10 byte out-of-order data is sent after *x* seconds. The *x* second value (step #7) is set to a value greater than *y* to detect reneging

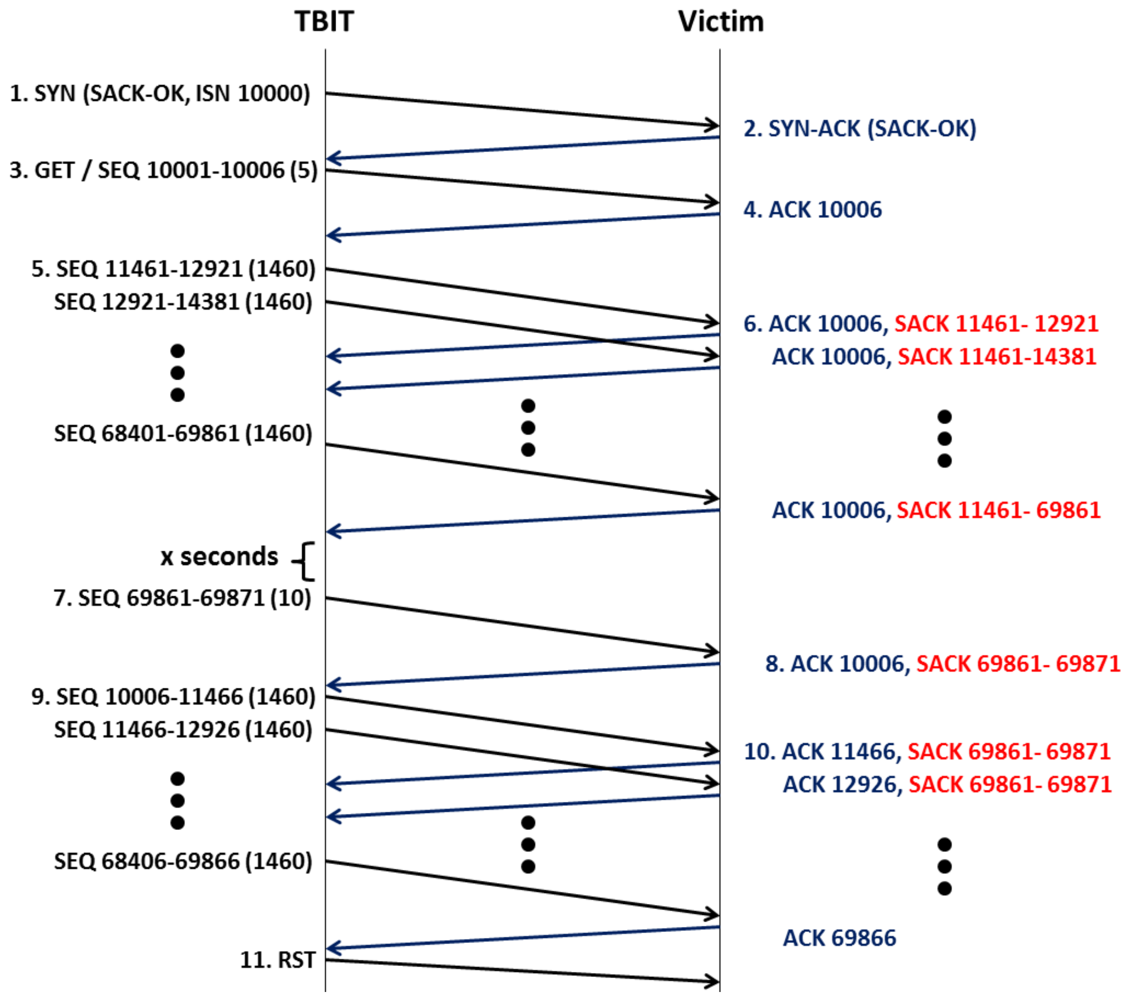

TBIT
Victim
1. SYN (SACK-OK, ISN 10000)
2. SYN-ACK (SACK-OK)
3. GET / SEQ 10001-10006 (5)
4. ACK 10006
5. SEQ 11461-12921 (1460)
SEQ 12921-14381 (1460)
6. ACK 10006, SACK 11461- 12921
ACK 10006, SACK 11461-14381
SEQ 68401-69861 (1460)
ACK 10006, SACK 11461- 69861
x seconds
7. SEQ 69861-69871 (10)
8. ACK 10006, SACK 69861- 69871
9. SEQ 10006-11466 (1460)
SEQ 11466-12926 (1460)
10. ACK 11466, SACK 69861- 69871
ACK 12926, SACK 69861- 69871
SEQ 68406-69866 (1460)
ACK 69866
11. RST

Figure 5.2: The CauseReneging TBIT Test with *m*=40 (step #5, #9)

using the response SACK (step #8). If that response SACKs only 10 bytes of out-of-order data as shown in Figure 5.2, one can conclude reneging occurred. To mimic a [RFC2018] conformant SACK implementation, *m+1* in-order segments are retransmitted (step #9), assuming a retransmission timeout value of *x* seconds. Recall that a TCP data sender is expected to discard SACK scoreboard at a retransmission timeout and retransmit bytes at the left edge of the window as specified in [RFC2018].

If reneging happens, ACKs (step #10) increase steadily after each in-order retransmission (step #9) as shown in Figure 5.2. Otherwise, the first ACK (step #10) acknowledges all the out-of-order data.

CauseReneg is a generic tool that can cause reneging on various victims (operating systems.) Only minimal changes are needed to run CauseReneg on different victims. The changes needed are setting the *m* value (step #5, #9) and *x* value (step #7) in the CauseReneging test, and the number of parallel TCP connections *n* that change dynamically from victim to victim. The values are determined by the victim's operating system, available main memory, and network buffers.

CauseReneg needs the ability to establish TCP connections to a victim. To establish a TCP connection, a port that is accessible (a server socket should be listening on the port and accept incoming TCP connections) is needed. Today, the majority of a machine's ports are blocked by firewalls for security purposes. Web servers on contrary are purposefully accessible. For that, CauseReneg is designed to attack a victim which deploys a web server (step #3 in Figure 5.2 sends the first 5 bytes of a HTTP GET request in-order). In our attempts to cause reneging, we installed Apache 2.2 in all potential victims. By default, Apache supports at most 256 TCP simultaneous connections. Recall that a busy web server with thousands TCP connections is a stronger candidate to renege. To simulate a busy web server, we increased the limit for simultaneous connections to 2000 which is enough to cause all victims to renege.

CauseReneg can attack victims regardless of their operating systems when a web server is running. Figure 5.3 presents an updated architecture for causing a remote host (victim) to renege. CauseReneg is used to attack various victims in a controlled network environment. A packet capture utility, tcpdump [Tcpdump], records TCP traffic between CauseReneg and a victim for later analysis. By analyzing the recorded TCP traffic, reneging instances can be detected via the RenegDetect tool detailed in Section 3.2.

Next we need to decide what victims to cause reneging. In Chapter 4, operating support for reneging is detailed for FreeBSD, Linux, Mac OS X, OpenBSD, Solaris and Windows. In Max OS X and OpenBSD, reneging is not possible by default. Therefore, we attempted to cause reneging on the following operating systems in which reneging is possible: FreeBSD 8.1, Linux 2.6.31, Solaris 11, Windows Vista and Windows 7. These systems are representative of popular operating systems with reneging support.

Four out of five operating systems (victims) are successfully reneged using CauseReneg tool. Unfortunately, we failed to cause a Linux 2.6.31 victim to renege. Linux implements dynamic right-sizing (DRS) where the rwnd dynamically changes based on the receiver's estimate of the sender's congestion window [Fisk 2001]. A data receiver increases rwnd when in-order data are received meaning the cwnd is increased. The initial advertised rwnd in Linux is 5840 bytes. CauseReneging sends only 5 bytes in-order data (step #3). Therefore, rwnd is not increased and limits CauseReneg to send 4380 (5840 – 1460) bytes of out-of-order data to the victim. In Linux, the receive buffer size is specified with *net.ipv4.tcp_rmem* sysctl with a default value of 87380 bytes. Recall from Section 4.3 that reneging in Linux is expected to happen when the memory allocated for receive buffer exceeds the memory limit available to the receive buffer. The minimum size of the receive buffer is specified with *net.ipv4.tcp_rmem* sysctl and is initialized to 4096 bytes. Apparently, sending

4380 bytes of out-of-order data was not enough to exceed the memory limit available to the receive buffer. Thus, DRS prohibited CauseReneg from sending more out-of-order data to trigger reneging. As a result, CauseReneg was unable to cause reneging in Linux.

The following sections, 5.2, 5.3 and 5.4, present consequences of reneging on FreeBSD, Solaris, and Windows victims, respectively. Section 5.5 concludes our efforts.

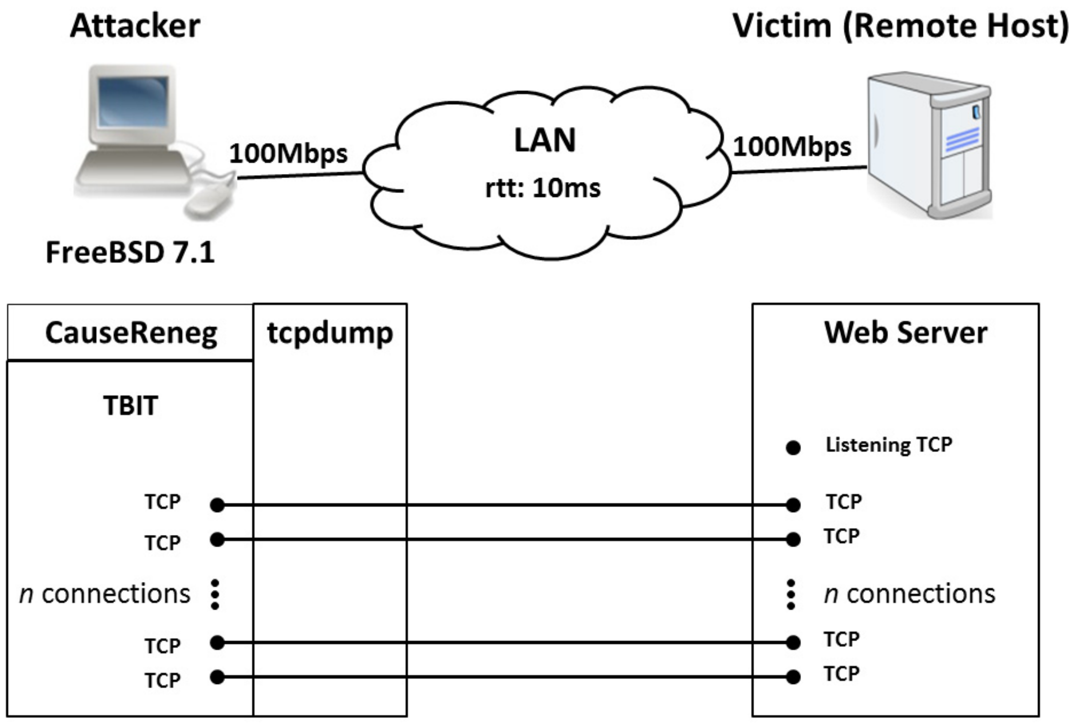


Figure 5.3: Causing a remote host to renege using CauseReneg

5.2 Causing a FreeBSD Host to Renege

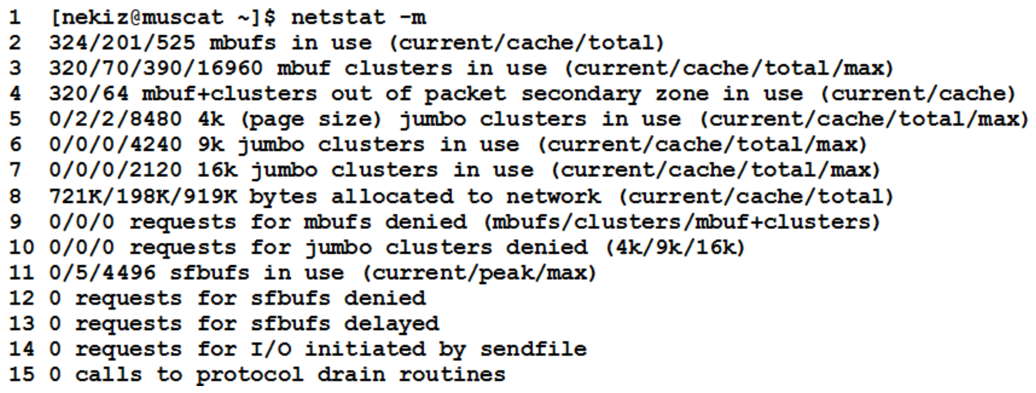
In this section, a FreeBSD 8.1 victim is reneged using CauseReneg. In Section 5.2.1, we first explain network buffers which are FreeBSD's structures to store TCP PDUs. In Section 5.2.2, two types of attacks are performed to cause a FreeBSD victim to renege. While one of the attacks crashes the operating system, the other one causes reneging. The circumstances of reneging are presented in Section 5.2.3.

5.2.1 Network Buffers

This section describes the network buffers used by FreeBSD to store network packets and FreeBSD's limits (sysctls) for TCP reassembly queues. In FreeBSD, all network packets are stored in structures known as mbuf(s) and mbuf clusters. An mbuf consists of a small internal buffer for data and a variable-sized header. While a network packet moves between different layers in the kernel, variable-size header changes as Ethernet, IP, and TCP headers are appended or removed from the mbuf header. The size of an mbuf is 256 bytes (specified in */usr/src/sys/sys/param.h*). If a TCP segment is small enough (less than 256 bytes), the segment's data are stored in the internal data buffer of an mbuf. If the segment is larger, either another mbuf is added to form an mbuf chain (implemented as linked list of mbufs) or external storage is associated with the mbuf [FreebsdImpl].

FreeBSD supplies a default type of external storage buffer called an mbuf cluster. The size of an mbuf cluster is machine dependent. Our victim is a FreeBSD 8.1 host where an mbuf cluster is 2048 bytes (defined in */usr/src/sys/sys/param.h*). The number of available external mbuf clusters can be read and modified via the *kern.ipc.nmbclusters* sysctl. Recall from Section 4.4 that a sysctl mechanism enables processes to get and set the kernel state in FreeBSD. Our victim has 16960 mbuf clusters by default shown in Figure 5.4 (line 3).

The *netstat -m* command reports the statistics recorded by the memory management routines for the available mbufs/mbuf clusters. Figure 5.4 shows an example output for the victim.



```
1  [nekiz@muscat ~]$ netstat -m
2  324/201/525 mbufs in use (current/cache/total)
3  320/70/390/16960 mbuf clusters in use (current/cache/total/max)
4  320/64 mbuf+clusters out of packet secondary zone in use (current/cache)
5  0/2/2/8480 4k (page size) jumbo clusters in use (current/cache/total/max)
6  0/0/0/4240 9k jumbo clusters in use (current/cache/total/max)
7  0/0/0/2120 16k jumbo clusters in use (current/cache/total/max)
8  721K/198K/919K bytes allocated to network (current/cache/total)
9  0/0/0 requests for mbufs denied (mbufs/clusters/mbuf+clusters)
10 0/0/0 requests for jumbo clusters denied (4k/9k/16k)
11 0/5/4496 sfbufs in use (current/peak/max)
12 0 requests for sfbufs denied
13 0 requests for sfbufs delayed
14 0 requests for I/O initiated by sendfile
15 0 calls to protocol drain routines
```

Figure 5.4: Network status output of a FreeBSD host

To gain insight on how network packets are stored in network buffers, variable number of consecutive out-of-order TCP segments of three different sizes (10, 100, or 1460 bytes) are sent to the victim. Table 5.1 presents numbers for the mbufs and mbuf clusters used to store the received out-of-order data. We conclude that irrespective of the segment size, an mbuf cluster (2048 bytes) is assigned to store an out-of-order TCP segment.

Table 5.1: Mbuf statistics for variable size out-of-order data for a single TCP connection

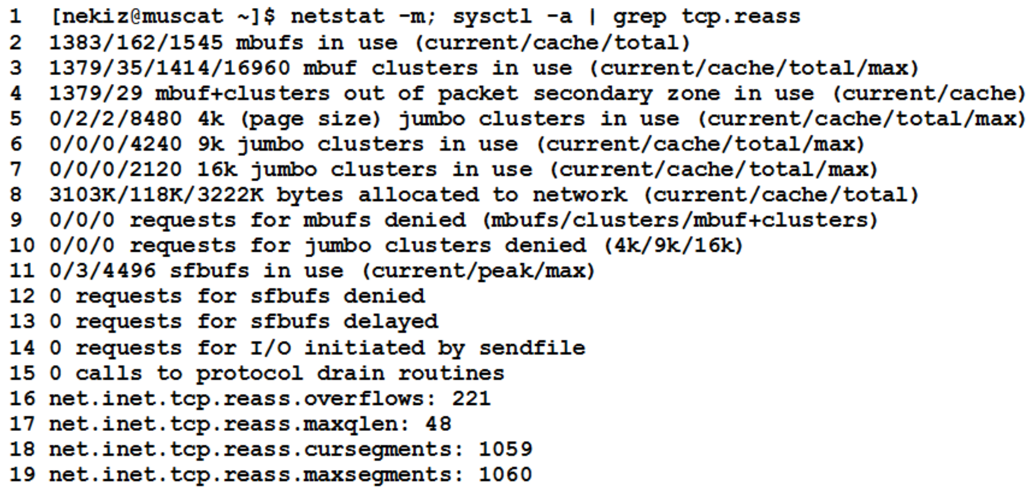
Segment size	Segments	Mbufs used	Mbuf clusters used
10 byte	1	1	1
10 byte	2	2	2
10 byte	4	4	4
100 byte	1	1	1
100 byte	2	2	2
100 byte	4	4	4
1460 byte	1	1	1
1460 byte	2	2	2
1460 byte	4	4	4

In FreeBSD, a TCP reassembly queue (or receive buffer) is implemented as an mbuf chain where data is stored in an external mbuf cluster. A reassembly queue is limited to store at most *net.inet.tcp.reass.maxqlen* ("Maximum number of TCP segments per individual Reassembly queue") out-of-order segments. The default value for *net.inet.tcp.reass.maxqlen* is 48 segments (48 * 1460 bytes = 70080 bytes). Using CauseReneg, a reassembly queue can be filled almost fully with out-of-order data since the victim's advertised TCP receive window of 65535 bytes is less than the reassembly queue limit.

Another sysctl, *net.inet.tcp.reass.maxsegments* ("Global maximum number of TCP segments in Reassembly queue"), defines the global limit for all segments in the all reassembly queues. FreeBSD assigns 1/16th of total mbuf clusters (16960) to *net.inet.tcp.reass.maxsegments* (1060). Once that limit is reached, arrived out-of-order segments are dropped. The *net.inet.tcp.reass.overflows* ("Global number of TCP Segment Reassembly Queue Overflows") sysctl reports the total number of dropped out-of-order segments. The *net.inet.tcp.reass.cursegments* ("Global number of TCP

Segments currently in Reassembly Queue") sysctl reports the total number of segments in all reassembly queues.

Now let us run a simple attack to the victim, using CauseReneg with *n*=32 parallel connections, to investigate the limits for the reassembly queues. The values for *m* and *x* are set to 40 and 200 seconds in the CauseReneging test (Figure 5.2), respectively. Figure 5.5 shows the statistics for mbuf/mbuf clusters along the TCP reassembly queue usage. CauseReneg sends a total of 1280 (*n*=32 * *m*=40) out-of-order segments to the victim where 1059 (line 18) of those segments are stored in the reassembly queues and 221 segments are dropped (line 16). When the maximum amount of out-of-order data are stored in the reassembly queues, the amount of memory allocated to network is 3222K. If reneging happened, FreeBSD would reclaim ~3M of main memory consumed by network buffers.



```
1  [nekiz@muscat ~]$ netstat -m; sysctl -a | grep tcp.reass
2  1383/162/1545 mbufs in use (current/cache/total)
3  1379/35/1414/16960 mbuf clusters in use (current/cache/total/max)
4  1379/29 mbuf+clusters out of packet secondary zone in use (current/cache)
5  0/2/2/8480 4k (page size) jumbo clusters in use (current/cache/total/max)
6  0/0/0/4240 9k jumbo clusters in use (current/cache/total/max)
7  0/0/0/2120 16k jumbo clusters in use (current/cache/total/max)
8  3103K/118K/3222K bytes allocated to network (current/cache/total)
9  0/0/0 requests for mbufs denied (mbufs/clusters/mbuf+clusters)
10 0/0/0 requests for jumbo clusters denied (4k/9k/16k)
11 0/3/4496 sfbufs in use (current/peak/max)
12 0 requests for sfbufs denied
13 0 requests for sfbufs delayed
14 0 requests for I/O initiated by sendfile
15 0 calls to protocol drain routines
16 net.inet.tcp.reass.overflows: 221
17 net.inet.tcp.reass.maxqlen: 48
18 net.inet.tcp.reass.cursegments: 1059
19 net.inet.tcp.reass.maxsegments: 1060
```

Figure 5.5: Statistics for mbuf and TCP reassembly queue size usage for 32 parallel TCP connections

5.2.2 Causing Reneging in FreeBSD

In this section, we explain two attacks to a FreeBSD victim using CauseReneg. The first attack crashes the victim accidentally while the second attack results in reneging.

As explained in Section 4.4, reneging in FreeBSD happens if the page replacement daemon (*vm_pageout*) invokes the *vm_pageout_scan()* function. When the available main memory goes low, and hard-coded or tunable paging thresholds are exceeded, *vm_pageout_scan()* is invoked to scan main memory to free some pages. If the memory shortage is severe enough, the largest process is also killed [Bruning 2005].

(A) To cause reneging, a variable number of parallel TCP connections are established to the victim using CauseReneg. The goal is to exhaust the main memory as much as possible to trigger reneging. Table 5.2 presents the initial memory statistics when *n* parallel TCP connections are established to the victim. Each TCP connection exhausts ~2.8MB of main memory. When more than ~250 active TCP connections are established, active virtual pages (the term used in FreeBSD for virtual pages of the running processes) stop increasing and the total memory allocated for the TCP connections is ~700MB. This amount of memory consumption is not enough to trigger reneging. The problem is due to Apache's initial *MaxClients* value (Maximum number of connections that will be processed simultaneously) that is set to 256 by default. As stated before, we expect reneging to happen at busy web servers serving thousands of TCP connections simultaneously. For this purpose, Apache is configured to support 2000 simultaneous connections.

Table 5.2: Memory usage statistics for *n* parallel TCP connections

n parallel TCP connections	Active virtual pages usage
1	3MB
2	6MB
4	11MB
8	22MB
16	45MB
32	90MB
200	558MB
300	701MB
400	701MB

Table 5.3 presents the updated memory statistics when *n* parallel TCP connections are established to the victim and Apache can serve up to 2000 connections simultaneously. With the ability to serve more TCP connections, the active virtual pages usage is increased beyond 700MB. While we expect reneging to happen with increased memory usage, the victim crashes instead of reneging! When the number of parallel connections exceeds 1241, the victim crashes with the following panic messages: (a) "Approaching the limit on PV entries, consider increasing either the *vm.pmap.shpgperproc* ("Page share factor per proc") or the *vm.pmap.pv_entry_max* ("Max number of PV entries") tunable" and (b) "panic: *get_pv_entry*: increase vm.pmap.shpgperproc". The panic messages are related to mapping of physical/virtual addresses of pages. To track the number of connections causing the victim crash easily, CauseReneg attacks the victim with the following configuration: *n*=1300, *m*=1, *x*=240 seconds. With this configuration, each TCP connection sends only 1 out-of-order segment to the victim. Figure 5.6 shows the statistics for TCP reassembly queue size and memory usage when 1241 parallel TCP connections (*net.inet.tcp.reass.cursegments*: 1059 (line 4) + *net.inet.tcp.reass.overflows*: 182 (line 2) = 1241) are established to the victim just before crashing.

```
[nekiz@muscat ~]$ sysctl -a | grep tcp.reass; vmstat
net.inet.tcp.reass.overflows: 182
net.inet.tcp.reass.maxqlen: 48
net.inet.tcp.reass.cursegments: 1059
net.inet.tcp.reass.maxsegments: 1060
 procs      memory      page                    disk   faults         cpu
 r b w     avm    fre   flt  re  pi  po    fr   sr ad0   in   sy   cs us sy id
 0 0 0   3533M   178M   592   0   0   0   388    0   0   11 1722 4539  1  3 97
```

Figure 5.6: Statistics for TCP reassembly queue size and memory usage for 1200+ parallel TCP connections

Table 5.3: Memory usage statistics for *n* parallel TCP connections (updated)

n parallel TCP connections	Active virtual pages usage
300	834MB
400	1127MB
500	1391MB
600	1687MB
700	1947MB
800	2267MB
900	2541MB
1000	2807MB
1100	3072MB
1200	3338MB
1241	3418MB

(B) In the second attack, to cause the page replacement daemon to call the *vm_pageout_scan()* function, a user process, shown in Figure 5.7, that consumes specified amount of main memory, is executed along with CauseReneg. If the memory shortage is severe enough due to the user process' excessive memory allocation and the victim goes low on main memory, the pageout replacement daemon is expected to kill the process using the largest memory (in that case the user process) and cause reneging.

For the second attack, CauseReneg attacks the victim with the following configuration: *n*=20, *m*=40, *x*=180 seconds. The attack is performed for two cases:

(B1) reneging is on (*net.inet.tcp.do_tcpdrain*=1), and (B2) reneging is off (*net.inet.tcp.do_tcpdrain*=0) at the victim.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]) {

  int numberOfBytes = 0;    // try to allocate user specified number of bytes argv[1]
  int bytes = 4096;         // default pagesize in FreeBSD
  int allocatedBytes = 0;   // total number of bytes allocated using malloc
  char *chPtr = NULL;

  if (argc == 2) {
    numberOfBytes = atoi(argv[1]);
  }

  while (allocatedBytes < numberOfBytes) {

    chPtr = (char *) malloc(bytes);
    chPtr = (char *) memset(chPtr, 0, bytes);

        // For each 100 MB print a message and sleep for 1 sec
    if ((allocatedBytes % (1024*1024*100)) == 0 ) {
      printf("Allocated 100MB! Sleep for 1 sec! Memory address: %p\n", chPtr);
      sleep(1);
    }

    allocatedBytes += bytes;

  }

  printf("Total allocated memory: %d (B) %2.f (KB) %.2f (MB)\n", allocatedBytes,
          ((double)allocatedBytes)/1024, ((double)allocatedBytes)/(1024*1024));

  return 0;
}
```

Figure 5.7: Main memory consumer program

If reneging happens, all the out-of-order data sent (step #5, Figure 5.2) are deleted from the receive buffers since FreeBSD employs global reneging. The SACK reply (step #8) for the 10 byte out-of-order data should be as 69861-69871 (10 bytes) as shown in Figure 5.2 for all TCP connections. ACKs (step #10) are expected to increase steadily after each in-order retransmission (step #9) along a SACK for the 10-byte out-of-order data.

If reneging does not happen, the out-of-order data should remain in the receive buffers. The SACK reply (step #10) for the 10 byte out-of-order data should be as 11461-69871 (58410 bytes). When the first in-order segment (10006-11466) is received (step #9) at the victim, the missing data between ACK and out-of-order data is received; hence an ACK with value 69871 should be returned (step #10.)

For the attacks, the architecture shown in Figure 5.3 is used. The victim (IP address: 128.4.30.23) has ~500MB physical memory, runs FreeBSD 8.1, and deploys Apache 2.2. During all attacks, the statistics for mbuf /mbuf clusters and TCP reassembly queue sizes are recorded. The TCP traffic between CauseReneg and the victim is also recorded for reneging analysis. The results are explained in the next Section 5.2.3.

5.2.3 Results

This section details the results of the attacks (A), (B1), (B2) described in the previous section. When the FreeBSD victim reneges, the following questions are answered to infer the consequences of reneging: (1) Does reneging help an operating system to resume its operation? (2) Can a reneged TCP connection complete a data transfer?

(A) Reneging does not happen, although memory consumption is high (3533MB), as shown in Figure 5.6. The reason is that the paging thresholds are not exceeded. If reneging happened, the operating system would reclaim ~3MB of main memory (recall from Figure 5.5 where all available space for out-of-order data is allocated). Since each TCP connection established consumes ~2.8MB, reclaimed memory would be consumed for the next TCP connection. Eventually, machine would crash anyways. I conclude that reneging does not benefit FreeBSD for such an attack.

For attacks (B1) and (B2), CauseReneg attacks the victim with the following configuration: *n*=20, *m*=40, *x*=180 seconds. Both attacks, (B1) (reneging is on) and (B2) (reneging is off), are performed in the following 7 step:

i. Start capturing the TCP traffic between the attacker and the victim on the attacker

ii. Record *netstat –m* output (mbuf statistics) and *sysctl –a* | *grep tcp.reass* output (reassembly queue size statistics) on the victim

iii. Attack the victim using CauseReneg

iv. Record *netstat –m* output (mbuf statistics) and *sysctl –a* | *grep tcp.reass* output (reassembly queue size statistics) on the victim

v. Run the user process (Figure 5.7) to allocate 2GB of main memory (*./a.out 2147483648*) on the victim

vi. Record *netstat –m* output (mbuf statistics) and sysctl –a | grep *tcp.reass* output (reassembly queue size statistics) on the victim

vii. Terminate capturing the TCP traffic between the attacker and the victim on the attacker after 5 minutes

Figure 5.8 shows the initial values of mbufs: 324 (line 2), mbuf clusters: 320 (line 3), and *net.inet.tcp.reass.cursegments*: 0 (line 18) for the attack (B1) before parallel TCP connections are established (step iii).

When the parallel connections are established, Figure 5.9 shows the updated statistics (step iv). The values for mbufs: 324 (initial) + 800 (out-of-order data) = 1124 (line 2), mbuf clusters: 320 (initial) + 800 (out-of-order data) = 1120 (line 3), and *net.inet.tcp.reass.cursegments*: 800 (line 18) are all consistent.

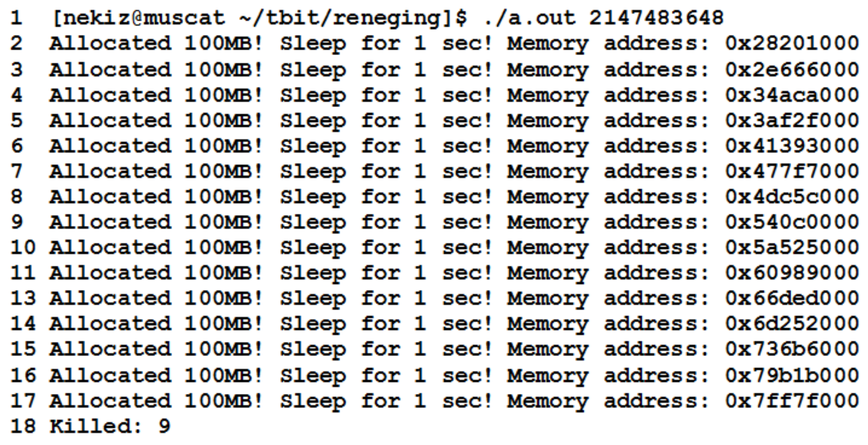
```
1  [nekiz@muscat ~/tbit/reneging]$ netstat -m; sysctl -a | grep tcp.reass
2  324/201/525 mbufs in use (current/cache/total)
3  320/70/390/16960 mbuf clusters in use (current/cache/total/max)
4  320/64 mbuf+clusters out of packet secondary zone in use (current/cache)
5  0/2/2/8480 4k (page size) jumbo clusters in use (current/cache/total/max)
6  0/0/0/4240 9k jumbo clusters in use (current/cache/total/max)
7  0/0/0/2120 16k jumbo clusters in use (current/cache/total/max)
8  721K/198K/919K bytes allocated to network (current/cache/total)
9  0/0/0 requests for mbufs denied (mbufs/clusters/mbuf+clusters)
10 0/0/0 requests for jumbo clusters denied (4k/9k/16k)
11 0/3/4496 sfbufs in use (current/peak/max)
12 0 requests for sfbufs denied
13 0 requests for sfbufs delayed
14 0 requests for I/O initiated by sendfile
15 0 calls to protocol drain routines
16 net.inet.tcp.reass.overflows: 0
17 net.inet.tcp.reass.maxqlen: 48
18 net.inet.tcp.reass.cursegments: 0
19 net.inet.tcp.reass.maxsegments: 1060
```

Figure 5.8: Step ii of causing reneging (reneging is on)

```
1  [nekiz@muscat ~/tbit/reneging]$ netstat -m; sysctl -a | grep tcp.reass
2  1124/166/1290 mbufs in use (current/cache/total)
3  1120/38/1158/16960 mbuf clusters in use (current/cache/total/max)
4  1120/32 mbuf+clusters out of packet secondary zone in use (current/cache)
5  0/2/2/8480 4k (page size) jumbo clusters in use (current/cache/total/max)
6  0/0/0/4240 9k jumbo clusters in use (current/cache/total/max)
7  0/0/0/2120 16k jumbo clusters in use (current/cache/total/max)
8  2521K/125K/2646K bytes allocated to network (current/cache/total)
9  0/0/0 requests for mbufs denied (mbufs/clusters/mbuf+clusters)
10 0/0/0 requests for jumbo clusters denied (4k/9k/16k)
11 0/3/4496 sfbufs in use (current/peak/max)
12 0 requests for sfbufs denied
13 0 requests for sfbufs delayed
14 0 requests for I/O initiated by sendfile
15 0 calls to protocol drain routines
16 net.inet.tcp.reass.overflows: 0
17 net.inet.tcp.reass.maxqlen: 48
18 net.inet.tcp.reass.cursegments: 800
19 net.inet.tcp.reass.maxsegments: 1060
```

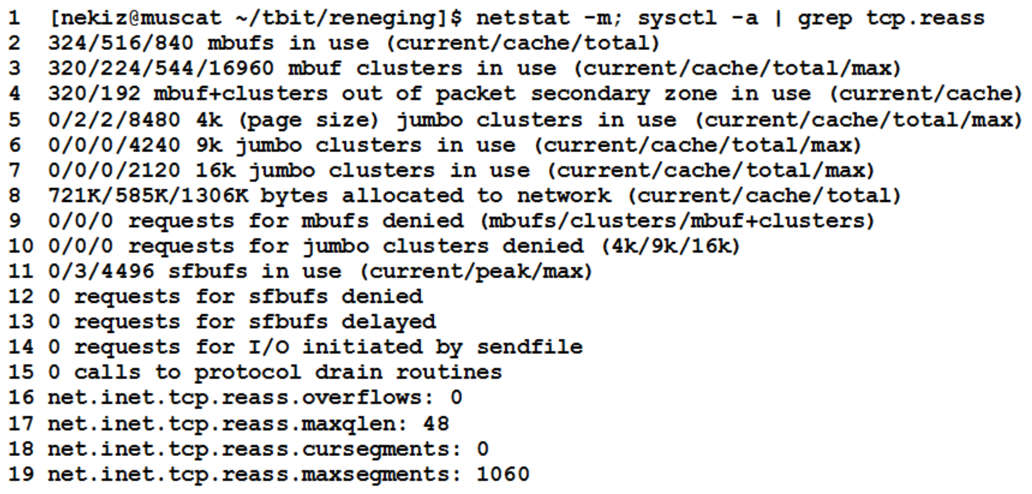
Figure 5.9: Step iv of causing reneging (reneging is on)

Figure 5.10 shows the execution of the user process (step v). FreeBSD allocates ~1.5GB of main memory to the user process before the user process is killed (line 18) by the page replacement daemon. At this point, reneging is expected to happen and the values for mbuf, mbuf clusters and *net.inet.tcp.reass.cursegments* should be the same as their initial values in Figure 5.8.



```
[nekiz@muscat ~/tbit/reneging]$ ./a.out 2147483648
Allocated 100MB! Sleep for 1 sec! Memory address: 0x28201000
Allocated 100MB! Sleep for 1 sec! Memory address: 0x2e666000
Allocated 100MB! Sleep for 1 sec! Memory address: 0x34aca000
Allocated 100MB! Sleep for 1 sec! Memory address: 0x3af2f000
Allocated 100MB! Sleep for 1 sec! Memory address: 0x41393000
Allocated 100MB! Sleep for 1 sec! Memory address: 0x477f7000
Allocated 100MB! Sleep for 1 sec! Memory address: 0x4dc5c000
Allocated 100MB! Sleep for 1 sec! Memory address: 0x540c0000
Allocated 100MB! Sleep for 1 sec! Memory address: 0x5a525000
Allocated 100MB! Sleep for 1 sec! Memory address: 0x60989000
Allocated 100MB! Sleep for 1 sec! Memory address: 0x66ded000
Allocated 100MB! Sleep for 1 sec! Memory address: 0x6d252000
Allocated 100MB! Sleep for 1 sec! Memory address: 0x736b6000
Allocated 100MB! Sleep for 1 sec! Memory address: 0x79b1b000
Allocated 100MB! Sleep for 1 sec! Memory address: 0x7ff7f000
Killed: 9
```

Figure 5.10: Step v of causing reneging (reneging is on)



```
[nekiz@muscat ~/tbit/reneging]$ netstat -m; sysctl -a | grep tcp.reass
324/516/840 mbufs in use (current/cache/total)
320/224/544/16960 mbuf clusters in use (current/cache/total/max)
320/192 mbuf+clusters out of packet secondary zone in use (current/cache)
0/2/2/8480 4k (page size) jumbo clusters in use (current/cache/total/max)
0/0/0/4240 9k jumbo clusters in use (current/cache/total/max)
0/0/0/2120 16k jumbo clusters in use (current/cache/total/max)
721K/585K/1306K bytes allocated to network (current/cache/total)
0/0/0 requests for mbufs denied (mbufs/clusters/mbuf+clusters)
0/0/0 requests for jumbo clusters denied (4k/9k/16k)
0/3/4496 sfbufs in use (current/peak/max)
0 requests for sfbufs denied
0 requests for sfbufs delayed
0 requests for I/O initiated by sendfile
0 calls to protocol drain routines
net.inet.tcp.reass.overflows: 0
net.inet.tcp.reass.maxqlen: 48
net.inet.tcp.reass.cursegments: 0
net.inet.tcp.reass.maxsegments: 1060
```

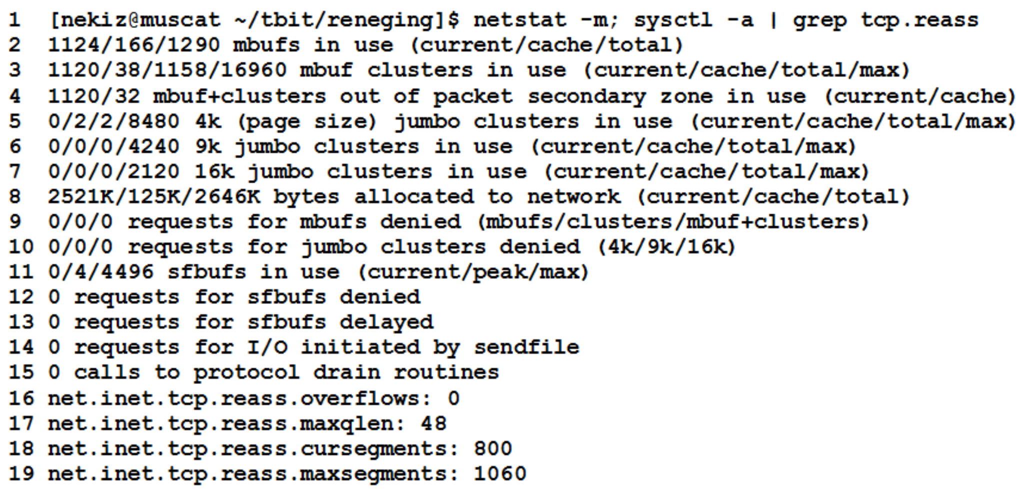
Figure 5.11: Step vi of causing reneging (reneging is on)

The output of (step vi) is shown in Figure 5.11. The number of mbufs: 324 (line 2) and mbuf clusters: 320 (line 3) are the same as their initial values. More importantly, *net.inet.tcp.reass.cursegments* is 0 (line 18) which concludes that reneging happens.

In Figure 5.11, network status output (*netstat -m*) reports the number of calls to the protocol drain routines (line 15) to be 0 even though reneging happens. We believe the functionality of *netstat* to report calls to protocol drain routines is not working properly and needs to be fixed.

Next, the attack (B2) is performed in 7 steps. When reneging is off, no out-of-order data are expected to be purged from the reassembly queues even though page replacement daemon invokes the *vm_pageout_scan()* function.

For the attack (B2), the outputs of step ii (Figure 5.8), iii, iv (Figure 5.9) and v (Figure 5.10) are all the same as of (B1).



```
1  [nekiz@muscat ~/tbit/reneging]$ netstat -m; sysctl -a | grep tcp.reass
2  1124/166/1290 mbufs in use (current/cache/total)
3  1120/38/1158/16960 mbuf clusters in use (current/cache/total/max)
4  1120/32 mbuf+clusters out of packet secondary zone in use (current/cache)
5  0/2/2/8480 4k (page size) jumbo clusters in use (current/cache/total/max)
6  0/0/0/4240 9k jumbo clusters in use (current/cache/total/max)
7  0/0/0/2120 16k jumbo clusters in use (current/cache/total/max)
8  2521K/125K/2646K bytes allocated to network (current/cache/total)
9  0/0/0 requests for mbufs denied (mbufs/clusters/mbuf+clusters)
10 0/0/0 requests for jumbo clusters denied (4k/9k/16k)
11 0/4/4496 sfbufs in use (current/peak/max)
12 0 requests for sfbufs denied
13 0 requests for sfbufs delayed
14 0 requests for I/O initiated by sendfile
15 0 calls to protocol drain routines
16 net.inet.tcp.reass.overflows: 0
17 net.inet.tcp.reass.maxqlen: 48
18 net.inet.tcp.reass.cursegments: 800
19 net.inet.tcp.reass.maxsegments: 1060
```

Figure 5.12: Step vi of causing reneging (reneging is off)

Figure 5.12 shows the memory statistics for the attack (B2) (step vi) after the user process is terminated by the page replacement daemon. The values for mbufs: 1124 (line 2), mbuf clusters: 1120 (line 3), and *net.inet.tcp.reass.cursegments*: 800 (line 18) are the same as of Figure 5.9 (step iv). Even though the *vm_pageout_scan()* is invoked and the user process is killed, the *tcp_drain()* is not called since reneging is disabled (off).

Both attacks (B1) and (B2) are analyzed using the RenegDetect tool detailed in Section 3.2. For (B1), RenegDetect successfully detects that all of the connections experience reneging. Figure 5.13 shows the tcpdump output of the last TCP connection (20th). The 40th out-of-order segment (68401-69861) is sent (lines 1, 2). In response, the victim sends an ACK (lines 3, 4) with SACK 11461-69861. After *x*=180 seconds, TBIT sends the 10 byte data (69861-69871) (lines 5, 6). The ACK for the 10 byte out-of-order data (lines 7, 8) has the SACK option 69861-69871; only for the 10 bytes sent giving the impression that reneging happens. When the first in-order data (10006-11466) are received (lines 9, 10), the victim returns an ACK 11466. This ACK strongly gives the impression that reneging happens. The next in-order data causes the victim to ACK 12926 (line 15). Consequently, ACKs are increased steadily after each in-order retransmission. This behavior concludes that reneging happens.

(B2) RenegDetect successfully detects that none of the connections experience reneging. Figure 5.14 shows the tcpdump output of the last TCP connection (20th). When the 40th out-of-order data (68401-69861) are received (lines 1, 2), an ACK with the SACK 11461-69861 is sent back (lines 3, 4). When the 10 byte out-of-order data are received, reply SACK is 11461-69871 as expected (lines 7, 8). Finally, when the first in-order data are received (lines 9, 10) at the victim, the gap in the reassembly queue is filled. As a result, ACK 69871 is sent back (lines 11, 12). This behavior concludes that reneging does not happen when reneging is turned off.

```
1  02:35:30.019170 IP 128.4.30.32.20019 > 128.4.30.23.80: Flags [P.], ack 1456902501,
2     win 21900, length 1460
3  02:35:30.019559 IP 128.4.30.23.80 > 128.4.30.32.20019: Flags [.], ack 2881110006,
4     win 65535, options [nop,nop,sack 1 {2881111461:2881169861}], length 0
5  02:38:30.030066 IP 128.4.30.32.20019 > 128.4.30.23.80: Flags [P.], ack 1456902501,
6     win 21900, length 10
7  02:38:30.030275 IP 128.4.30.23.80 > 128.4.30.32.20019: Flags [.], ack 2881110006,
8     win 65535, options [nop,nop,sack 1 {2881169861:2881169871}], length 0
9  02:38:30.049606 IP 128.4.30.32.20019 > 128.4.30.23.80: Flags [P.], ack 1456902501,
10    win 21900, length 1460
11 02:38:30.050009 IP 128.4.30.23.80 > 128.4.30.32.20019: Flags [.], ack 2881111466,
12    win 64240, options [nop,nop,sack 1 {2881169861:2881169871}], length 0
13 02:38:30.069584 IP 128.4.30.32.20019 > 128.4.30.23.80: Flags [P.], ack 1456902501,
14    win 21900, length 1460
15 02:38:30.069998 IP 128.4.30.23.80 > 128.4.30.32.20019: Flags [.], ack 2881112926,
16    win 64240, options [nop,nop,sack 1 {2881169861:2881169871}], length 0
```

Figure 5.13: Tcpdump output of a TCP connection from causing reneging (reneging is on)

```
1  02:59:47.237359 IP 128.4.30.32.20019 > 128.4.30.23.80: Flags [P.], ack 4179487467,
2     win 21900, length 1460
3  02:59:47.237805 IP 128.4.30.23.80 > 128.4.30.32.20019: Flags [.], ack 2881110006,
4     win 65535, options [nop,nop,sack 1 {2881111461:2881169861}], length 0
5  03:02:47.242328 IP 128.4.30.32.20019 > 128.4.30.23.80: Flags [P.], ack 4179487467,
6     win 21900, length 10
7  03:02:47.242530 IP 128.4.30.23.80 > 128.4.30.32.20019: Flags [.], ack 2881110006,
8     win 65535, options [nop,nop,sack 1 {2881111461:2881169871}], length 0
9  03:02:47.261902 IP 128.4.30.32.20019 > 128.4.30.23.80: Flags [P.], ack 4179487467,
10    win 21900, length 1460
11 03:02:47.262392 IP 128.4.30.23.80 > 128.4.30.32.20019: Flags [.], ack 2881169871,
12    win 5835, length 0
```

Figure 5.14: Tcpdump output of a TCP connection from causing reneging (reneging is off)

A FreeBSD victim is reneged with the attack (B1). Now, we answer the following questions to gain insight to the consequences of reneging: (1) Does reneging help an operating system to resume its operation? (2) Can a reneged TCP connection complete a data transfer?

(1) After the attack (B1), the FreeBSD victim continues to resume normal operation. As stated before, only ~3MB of main memory (the maximum amount possible for the victim) used for the network buffers is reclaimed back to the operating system. Since the memory shortage, caused by the attack, is severe, the largest process (~1.5GB) is killed. I believe the amount of main memory used for network buffers is negligible compared to the process using the most memory. Reneging alone does not seem to help an operating system resume normal operation, and the reassembly queues' memory was wastefully purged. The attack (B2), where reneging was disabled for the second attack, demonstrated that FreeBSD could resume normal operation without reneging. Therefore, I argue that the current handling of reneging is wrong and reneging should be turned off by default in FreeBSD as in Mac OS X.

To answer (2), we need to test if the TCP data senders do implement tolerating reneging properly as specified in [RFC2018]. Recall that a TCP sender needs to discard its SACK scoreboard at a retransmission timeout and start sending bytes at the left edge of the window. Otherwise, reneging may cause a data transfer to stall (fail).

FreeBSD employs a global reneging strategy that all TCP connections with out-of-order data are reneged. If TCP connections with out-of-order data from various TCP data senders are established to the FreeBSD victim before the (B1) attack, those TCP connections would renege too. To test if [RFC2018] conformant tolerating reneging is implemented, a 5MB file is transferred using secure shell (ssh) to the FreeBSD victim from various operating systems listed in Table 5.4. To create out-of-order data for those transfers, Dummynet is configured on the FreeBSD victim to drop 15-20% of the TCP PDUs. The traffic between a TCP data sender and the FreeBSD victim is recorded for reneging analysis. Once a data transfer starts, the FreeBSD victim is reneged using the attack (B1) and we observe if the file transfer experiencing reneging can be completed. In all data transfers, reneging is detected by analyzing the recorded traffic using the RenegDetect tool. We confirm that all of the TCP data senders in Table 5.4 complete the data transfer successfully. In conclusion, [RFC2018] conformant tolerating reneging is implemented in all TCP stacks tested.

Table 5.4: Testing [RFC2018] conformant TCP data senders

Operating System	Transfer Completed	Reneging
FreeBSD 8.0	yes	yes
Linux 2.6.24	yes	yes
Mac OS X 10.8.0	yes	yes
NetBSD 5.0.2	yes	yes
OpenBSD 4.8	yes	yes
OpenSolaris 2009.06	yes	yes
Solaris 11	yes	yes
Windows XP	yes	yes
Windows Vista	yes	yes
Windows 7	yes	yes

5.3 Causing a Solaris Host to Renege

In this section, a Solaris 11 victim is reneged and the consequences of reneging are detailed. First, Section 5.3.1 details the attack to cause reneging. Next, in Section 5.3.2, the consequences of reneging in Solaris are presented.

5.3.1 Causing Reneging in Solaris

The circumstances to cause a Solaris host to renege are detailed in Section 4.6. If out-of-order data sits in the TCP reassembly queue for at least 100 seconds (the default reassembly timer timeout value), a Solaris receiver would renege and purge the entire reassembly queue. Reneging, in such case, protects the operating system against DoS attacks.

In the CauseReneging test (see Figure 5.2), *m* out-of-order segments are sent (step #5) to the victim. Later, 10 byte out-of-order data are sent (step #7) after *x* seconds to check if reneging happened. Reneging in Solaris is expected to happen 100 seconds after the arrival of out-of-order data (step #5). To force the reassembly queue timer to expire, *x* should be set to a value > 100 seconds. The number of parallel connections (*n*) and out-of-order segments (*m*) can be set arbitrarily since reneging in Solaris only depends on *x*. CauseReneg attacks the victim with the following configuration: *n*=20, *m*=40, *x*=180 seconds. The value for *m* is set to 40 purposefully to explain reneging using Figure 5.2. With this configuration, reneging is expected to happen before 10 byte out-of-order data are sent (step #7).

If reneging happens, the out-of-order data sent (11461-69861) (step #5) are removed from the reassembly queues of all the 20 parallel TCP connections before (step #7.) The reply SACK for 10 byte out-of-order data should be 69861-69871 (step #8). Consequently, ACKs (step #10) should be increased steadily after each in-order data retransmission (step #9.)

If reneging does not happen, the reply SACK (step #8) for the 10 byte out-of-order data should be 11461-69871 (58410 bytes). In (step #9), the first in-order data should fill the gap between the ACK and out-of-order data, and increase ACK to 69871 (step #10.)

For the attack, the architecture shown in Figure 5.3 is used. Solaris 11 is installed on an Ubuntu 9.10 host (Linux 2.6.24) using Oracle's VirtualBox virtualization software [Virtualbox]. The victim has 1024MB physical memory, runs Solaris 11, and deploys Apache 2.2. The TCP traffic between CauseReneg and the victim is recorded for latter analysis. The result of the attack is explained in the next Section 5.3.2.

5.3.2 Results

Data reneging happens when the TCP reassembly queue timer expires after 100 seconds (the default value of reassembly queue timer) for the out-of-order data sent (step #5) in CauseReneging test, shown in Figure 5.2. A tcpdump output of the 6th parallel connection is shown in Figure 5.15. Please refer to Figure 5.2 for references using (step #p) and Figure 5.15 for references using (lines p). The 40th out-of-order data (68401-69861) (step #5) from CauseReneging is shown (lines 1, 2). The reply SACK (step #6) acknowledges all the out-of-order data received (11461-69861) (lines 3, 4). After *x*=180 seconds, 10 byte out-of-order data (69861-69871) are sent (step #7) (lines 5, 6). The reply SACK (step #8) demonstrates evidence of reneging since only 10 out-of-order bytes are selectively acknowledged (69861-69871) (lines 7, 8). After the in-order received data (step #9), the victim's ACKs (step #10) are steadily increased as expected.

```
1 21:06:43.572124 IP 128.4.30.32.20005 > 128.4.30.29.80:
2     P 2881168401:2881169861(1460) ack 128403387 win 21900
3 21:06:43.572768 IP 128.4.30.29.80 > 128.4.30.32.20005:
4     . ack 2881110006 win 64240 <nop,nop,sack 1 {2881111461:2881169861}>
5 21:09:43.579360 IP 128.4.30.32.20005 > 128.4.30.29.80:
6     P 2881169861:2881169871(10) ack 128403387 win 21900
7 21:09:43.579739 IP 128.4.30.29.80 > 128.4.30.32.20005:
8     . ack 2881110006 win 64240 <nop,nop,sack 1 {2881169861:2881169871}>
```

Figure 5.15: Tcpdump output of a TCP connection from causing reneging attack on Solaris 11

The traffic recorded during the attack is analyzed using RenegDetect tool explained in Section 3.2.The RenegDetect detects that all the connections (20) used in the attack experienced reneging.

We believe reneging in Solaris is used as a mechanism to protect against DoS attacks. A TCP sender is expected to retransmit lost segments *r* times (for example, *TcpMaxDataRetransmissions* in Windows Server 2003 defines *r*=5 by default.) After *r* retransmissions, a TCP sender would terminate a TCP connection. The loss recovery period takes at most 1-2 minutes (assuming back to back timeouts, an initial retransmission timeout value (RTO) of 1 second, and *r*=5.) When out-of-order data sit in the reassembly queue for at least 100 seconds (the default reassembly queue timer value) at the Solaris receiver, one can infer that either the TCP sender terminated the connection or the host is under a DoS attack where the out-of-order data intentionally exhaust host's resources. Therefore, cleaning the reassembly queue seems a useful mechanism in both cases. Instead of just releasing the out-of-order data, a better option would be to RESET the connection when reneging is caused by either a terminated TCP connection (due to loss recovery) or a DoS attack. With that change, all the resources used for the TCP connection are released, therefore better utilized.

5.4 Causing Windows Hosts to Renege

In this section, Windows Vista and 7 victims are reneged, and the consequences of reneging in Windows systems are detailed. Section 5.4.1 details the attacks to cause reneging and Section 5.4.2 presents the consequences of reneging in Windows.

5.4.1 Causing Reneging in Windows

Reneging support for Microsoft's Windows is detailed in Section 4.1. Dave MacDonald, the author of Microsoft Windows 2000 TCP/IP Implementation Details [MacDonald 2000], stated that Vista and its successors implement reneging as a protection mechanism against DoS attacks. Reneging happens when the memory consumption of total TCP reassembly data in relation to the global memory limits is significant. To investigate the consequences of reneging, CauseReneg attacks a Windows victim (either Vista or 7) by increasing the number of parallel connections to make the memory consumption of the total reassembly data so significant that reneging is triggered.

The initial advertised window (rwnd) in both Window's Vista and 7 is 64240 bytes corresponding to 44 * 1460 byte TCP PDUs. Based on the initial rwnd, the m value, the number of out-of-order segments, in the CauseReneging test (step #5) is set to 43 to fill each reassembly queue almost fully with out-of-order data. In the attacks, the number of parallel connections established to the victim (n) and the x seconds (step #7) values in the CauseReneging test are variable.

For the attacks, the architecture shown in Figure 5.3 is used. Windows Vista and 7 operating systems are installed on an Ubuntu 9.10 host (Linux 2.6.24) using Oracle's VirtualBox virtualization software [Virtualbox]. The Vista victim has 2GB physical memory whereas the Windows 7 victim has 1GB memory. Both victims deploy Apache 2.2 which can serve 2000 simultaneous TCP connections. The TCP traffic between CauseReneg and the victims is recorded for latter analysis. The results of the attacks are explained in the next Section 5.4.2.

5.4.2 Results

First, the Vista victim is attacked by CauseReneg using the following configuration: n=variable, m=43, x=200 seconds. Table 5.5 presents the results of the attacks. When the parallel connections established are 100 or 200, reneging does not happen. When 300 parallel connections are established, only the last 33 connections renege. When the parallel connections established are 400, a similar behavior happens. The first 267 connections do not renege but the last 133 connections do renege. This behavior implies that the memory consumption of total reassembly data in relation to the global memory limit is considered significant in Vista when the out-of-order data in the reassembly queue is at least ~16MB (267 (parallel connections) * 43 (out-of-order segments) * 1460 bytes). To verify that the global memory limit for reneging is ~16MB, another attack is performed with the configuration: n=600, m=20, x=200 seconds. With this configuration, only ~half of the rwnd is filled with out-of-order data. The observed behavior is consistent: only the last 25 of 600 connections renege, and the memory allocated to out-of-order data before reneging happens is again ~16MB (575 (parallel connections) * 20 (out-of-order segments) * 1460 bytes).

Table 5.5: CauseReneg attack to a Vista victim with variable number of parallel connections

n parallel TCP connections	Reneging
100	No
200	No
300	Yes (33 connections renege)
400	Yes (133 connections renege)

Next, we test if Windows implements a reassembly queue timer similar to Solaris 11. For that purpose, CauseReneg attacks the Vista victim using the following configuration: *n*=300, *m*=43, *x*=variable seconds. The attacks are performed for *x* = {30, 40, 50, 100, 200 seconds}. The same behavior is observed in the all attacks: only the last 33 of 300 connections renege.

Last, the Windows 7 victim is attacked by CauseReneg using the following configuration: *n*=variable, *m*=43, *x*=200 seconds. Table 5.6 presents the results of the attacks. When the parallel connections established are 100, reneging does not happen. When 200 parallel connections are established, the first 133 connections do not renege but the last 67 connections renege. When the parallel connections established are 300, the first 133 connections do not renege but the last 167 connections renege. This behavior implies that the memory limit for the reassembly queue for the Windows 7 victim is ~8MB (133 (parallel connections) * 43 (out-of-order segments) * 1460 bytes). Recall that the Vista victim has a physical memory of 2GB whereas the Windows 7 victim's memory is 1GB. The memory limit used for the reassembly data to trigger reneging in both systems is ~0.78% of the physical memory and seems to scale with the physical memory.

Table 5.6: CauseReneg attack to a Windows 7 victim with variable number of parallel connections

n parallel TCP connections	Reneging
100	No
200	Yes (67 connections renege)
300	Yes (167 connections renege)

In conclusion, Windows Vista+ supports reneging as a protection mechanism against DoS attacks, reneges when the memory threshold for reassembly data is reached, and resumes normal operation after reneging.

5.5 Conclusion

We detailed a tool, CauseReneg, to cause a victim to renege in Section 5.1. CauseReneg achieves its goal by exhausting a victim’s resources by sending out-of-order data using multiple TCP connections. To document the consequences of reneging, CauseReneg attacks various victims deploying popular operating systems with reneging support such as FreeBSD, Linux, Solaris, and Windows.

For FreeBSD, two attacks are performed to a victim. The first one caused the victim to crash and the second one to renege. In both attacks, the available main memory is largely consumed by out-of-order data to trigger reneging. In the first attack, the page replacement daemon does not invoke the reneging routines, probably due to low paging activity, even though the total memory used for 1240+ parallel TCP connections is ~3.3GB (victim has 500MB of physical memory.) In this attack, the victim crashes and reneging does not help the operating system to resume normal operation. In the second attack, a user process allocating 2GB of main memory is used along CauseReneg to cause high paging activity and reneging. This time, the page replacement daemon invokes drain routines, and TCP reneges. All of the reassembly queues of active TCP connections are purged to reclaim main memory to FreeBSD, and the process using the largest memory allocation is terminated by the page replacement daemon.

Initially, it was thought that an operating system starving for main memory would eventually crash. Our first attack is such an example. In the second attack, when the paging activity is high and the available memory is low, reneging happens in addition to the largest process getting killed. This time, FreeBSD resumes normal operation. The maximum amount of memory that can be allocated to reassembly queues by reneging is limited to ~3MB (0.6% of the physical memory) for the victim attacked. That amount of memory seems negligible compared to the process using the most memory. Reneging alone does not seem to help FreeBSD resume normal operation, and the reassembly queues' memory was wastefully purged. The attack (B2), where reneging was disabled for the second attack, demonstrated that FreeBSD could resume normal operation without reneging. Therefore, I argue that reneging support should be turned off by default in FreeBSD as in Mac OS X.

A reneging TCP connection can complete a data transfer only if the TCP sender implements tolerating reneging as specified in [RFC2018]. Otherwise, the data transfer would fail (stall). To tolerate reneging, a TCP sender is expected to clear its SACK scoreboard at a retransmission timeout (RTO) and retransmit bytes from the left edge of the window. To validate this behavior, we transferred a 5MB file from various operating systems, listed in Table 5.4, to a reneging FreeBSD victim. Our experiment confirms that all the operating systems tested complete a data transfer after the connection experiences reneging.

FreeBSD employs global reneging as explained in Section 4.4. When reneging happens, all the reassembly queues are cleared. On the other hand, Linux and Solaris employ local reneging as explained in Sections 4.3 and 4.6, respectively. In local reneging, only the individual connections are reneged.

Global reneging is easy to implement. A single reneging function is defined, and no bookkeeping is needed. The reneging function is invoked for all active TCP connections when reneging is needed. The disadvantage with global reneging is that if the memory required by the operating system to resume normal operation is less than the total memory allocated for the reassembly queues, some TCP connections are unnecessarily penalized.

Local reneging, on the other hand, is more complex to implement, and requires bookkeeping for each TCP connection and global memory pools. As a connection progresses, the amount of allocated receive buffer space is recorded as data is appended/removed from the receive buffer. In local reneging, only those connections exceeding memory limits experience reneging. Therefore, local reneging is fairer compared to global reneging. If I were to implement reneging, I would choose local reneging.

A Solaris 11 victim is reneged in Section 5.3. In Section 5.4, Windows Vista and 7 victims are reneged. Both operating systems, Solaris and Windows, use reneging as a protection mechanism against DoS attacks. The difference between Solaris and Windows is that the former uses a uses a reassembly queue timer to renege whereas the latter uses a memory threshold for the out-of-order data for the same purpose.

In Solaris, when out-of-order data sit in the reassembly queue for at least 100 seconds, reneging happens. It can be inferred that the connection is either terminated due to loss recovery or exhausts resources intentionally (a DoS attack.) In both cases, instead of reneging, terminating the connection with RESETs seems to be a better option. RESETing would release all of the resources held.

In Windows, reneging happens when the memory allocated for out-of-order data exceeds the memory threshold available for the reassembly data. This threshold appears to be ~0.78% of the available physical memory. The current reneging implementation has a potential problem. The out-of-order data that cause reaching the threshold are not reneged. Instead, the out-of-order data received afterwards are reneged. Were an attacker to find out the memory threshold (as we did in Section 5.4.1) and only send that amount of out-of-order data, all future connections experiencing losses and receiving out-of-order data afterwards would renege. A TCP data sender would not retransmit SACKed data until a retransmission timeout (RTO) [RFC2018]. In such a case, losses would be recovered with RTOs resulting in increased transfer times (lower throughput.) The quality of service, data transfer times for legitimate users, would be reduced. That type of an attack can be referred as reduction of service (RoS) attack. We believe that a RoS attack would be harder to detect compared to a DoS attack since the service provided in not interrupted but slowed.

When we compare reneging in Solaris vs. Windows, Solaris's approach seems to be a better protection mechanism: only the DoS connections are penalized. An important disadvantage of Solaris's implementation is using a timer. Managing a TCP timer is an expensive operation.

In summary, reneging is caused for FreeBSD, Solaris, and Windows victims using CauseReneg tool. The consequences of reneging are detailed for those systems. When an operating system (e.g. FreeBSD) is starving for memory, reneging alone cannot help the system to resume normal operation. Therefore, I argue that reneging support should be turned off for systems employing that type of reneging. Reneging in Solaris and Windows protects the system against DoS attacks. I argue that type of protection is essential to operating systems but I believe that a better approach would be to RESET the connection under the attack instead of reneging.

Chapter 6

PRIOR COLLOBORATIVE RESEARCH

Prior to the research contributions of this dissertation, I have been involved with ns-2' [Ns-2] SCTP module for more than three years. Currently, I maintain the SCTP module which was developed in UD's Protocol Engineering Lab (PEL). I have been involved with two completed projects to support past PhD student Preethi Natarajan. The activities I have been involved include running ns-2 experiments, fixing bugs and adding new extensions to the ns-2' SCTP module. The next two sections present my contributions to Non-Renegable Selective Acknowledgments (NR-SACKs) and Concurrent Multipath Transfer (CMT)/Potentially Failed (PF) projects.

6.1 NR-SACKs

In both TCP and SCTP, selectively acknowledged (SACKed) out-of-order data is implicitly renegable; that is, the receiver can later discard SACKed data. The possibility of reneging forces the transport sender to maintain copies of SACKed data in the send buffer until they are cumulatively ACKed.

In [Natarajan 2008b], we investigated the situation where all out-of-order data are non-renegable, such as when the data has been delivered to the application, or when the receiver simply never reneges either by agreement or if the user has explicitly turned off reneging using sysctl controls. Using ns-2 simulations, we showed that SACKs result in inevitable send buffer wastage, which increases as the frequency of loss events and loss recovery durations increases. We introduced a fundamentally new ACK mechanism, Non-Renegable Selective Acknowledgments (NR-SACKs), for SCTP. Using NR-SACKs, an SCTP receiver explicitly identifies some or all out-of-order data as being non-renegable, allowing the data sender to free up send buffer sooner than if the data were only SACKed. We compared and showed that NR-SACKs enable efficient utilization of a transport sender's memory. We further investigated the effects of using NR-SACKs in Concurrent Multipath Transfer (CMT). Using ns-2 simulations, we showed that NR-SACKs not only reduce transport sender's memory requirements, but also improve throughput in CMT.

In [Yilmaz 2010], we extended the investigation of the throughput improvements that NR-SACKs can provide, particularly when all out-of-order data are non-renegable. Using ns-2 simulations, for various loss conditions and bandwidth-delay combinations, we showed that the throughput observed with NR-SACKs is at least equal and often better than the throughput observed with SACKs. We introduced "region of gain" which defines for a given bandwidth, delay, and send buffer size combination, what interval of loss rates results in significant throughput improvement when NR-SACKs are used instead of SACKs. In both SCTP and CMT, NR-SACKs provided greater throughput improvement as the send buffer size decreases, and as the end-to-end delay decreases. Provided that the bandwidth-delay product (BDP) $\geq$ send buffer size, additional bandwidth does not affect NR-SACKs' throughput improvements for either SCTP or CMT. For BDPs < send buffer size, the throughput improvement using NR-SACKs decreases as the BDP decreases. We also presented details of our NR-SACK implementation in FreeBSD, and analyzed NR-SACKs vs. SACKs over a Dummynet-emulated network [Dummynet] using our FreeBSD SCTP stack. Note: Preethi Natarajan and I added the support for NR-SACKs in the ns-2' SCTP module, and Ertugrul Yilmaz implemented NR-SACKs in the FreeBSD SCTP stack.

I am the co-author of [Natarajan 2008b], [Natarajan 2009 (b)] and [Yilmaz 2010]. In the NR-SACK project, I ran the experiments to compare NR-SACK vs. SACK on various network topologies, path characteristics and loss models. Preethi and I added support for NR-SACKs for SCTP, CMT and CMT-PF in the SCTP module. I also added the support to track send buffer utilization for both NR-SACKs and SACKs. I included new validation tests for both NR-SACK and SACK, and submitted a patch (SCTP module 3.8 released with ns-2.35) to the main trunk of ns-2 which adds support for NR-SACKs.

Varun Notibala and I also implemented viewing and graphing NR-SACKs data transfers in the Wireshark network protocol analyzer tool [Wireshark].

6.2 Concurrent Multipath Transfer (CMT)/Potentially Failed (PF)

Concurrent Multipath Transfer (CMT) uses SCTP's multihoming feature to distribute data across multiple end-to-end paths in a multihomed SCTP association [Iyengar 2006]. Since data are sent simultaneously on different paths, data reordering is inevitable. The author investigated the negative effects of data reordering and introduced new algorithms to deal with data reordering problem.

[Iyengar 2007] explored the performance of CMT in the presence of a constrained receive buffer and investigated the receive buffer blocking problem observed in CMT transfers. Different retransmission policies were evaluated under various bounded receive buffer sizes. The authors showed that the receive buffer blocking cannot be eliminated but can be reduced with a well-chosen retransmission policy.

Janardhan Iyengar extended ns-2' SCTP module to support CMT and implemented CMT in the FreeBSD SCTP stack.

[Natarajan 2006] investigated CMT's throughput degradation caused by receive buffer blocking during complete and/or short-term network failures. To improve CMT's performance during a failure, a new state for each destination called the "Potentially-Failed" (PF) state and a retransmission policy that takes into account the PF state was introduced. Using ns-2 simulations, CMT-PF was evaluated, and throughput improvements were shown over CMT in failure-prone networks.

[Natarajan 2008a] completed the evaluation of CMT vs. CMT-PF. Using ns-2 simulations we showed that CMT-PF performs on par or better than CMT during more aggressive failure detection thresholds than recommended by [RFC4960]. We also examined whether the modified sender behavior in CMT-PF degrades performance during non-failure scenarios. Our evaluations considered: (1) realistic loss model with symmetric and asymmetric path loss, (2) varying path RTTs. We found that CMT-PF performs as well as CMT during non-failure scenarios, and interestingly, outperforms CMT when the paths experience asymmetric receive buffer blocking conditions. We recommended that CMT be replaced by CMT-PF in future CMT implementations and RFCs.

In [Natarajan 2009], we confirmed our simulations results using FreeBSD implementations of CMT and CMT-PF.

Preethi Natarajan added the support for CMT-PF in the SCTP module of ns-2 and Joe Szymanski implemented CMT-PF in the FreeBSD SCTP stack.

I am the co-author of [Natarajan 2008a] and [Natarajan 2009] and I have been involved with the following activities for this research project. I ran the experiments to compare CMT vs. CMT-PF on various network topologies, path characteristics and loss models. I discovered several bugs in ns-2' SCTP module for CMT and CMT-PF and then fixed them. I wrote new validation tests for both CMT and CMT-PF, and submitted a patch (SCTP module 3.7 released with ns-2.32) to the main trunk of ns-2 which adds support for CMT-PF. I was also involved with debugging the FreeBSD's CMT-PF code.

Chapter 7

CONCLUSIONS & FUTURE WORK

7.1 Conclusions

Reneging occurs when a data receiver SACKs data, and later discards these data from its receive buffer prior to delivering these data to the receiving application. TCP is designed to tolerate reneging. Specifically [RFC2018] states that: "*The SACK option is advisory, in that, while it notifies the data sender that the data receiver has received the indicated segments, the data receiver is permitted to later discard data which have been reported in a SACK option*". Reneging may happen when an operating system needs to recapture previously allocated receive buffer memory for another process, say to avoid deadlock.

Because TCP is designed to tolerate possible reneging by a data receiver, a TCP data sender must keep copies of all transmitted data segments in its send buffer, even SACKed data, until cumulatively ACKed. If reneging does happen, a copy of the reneged data exists and can be retransmitted to complete the reliable data transfer. Inversely if reneging does not happen, SACKed data are unnecessarily stored in the send buffer until cumulatively ACKed.

I argue that this design assumption to tolerate reneging is wrong. To support my argument, this dissertation investigated (1) the instances, (2) causes and (3) effects of TCP reneging in today's Internet.

(1) To document the instances and the frequency of TCP reneging in Internet traces, we proposed a mechanism to detect reneging instances. The proposed

mechanism is based on how an SCTP data sender infers reneging. A state of the receive buffer is constructed at an intermediate router and updated as new acks are monitored. When an inconsistency occurs between the state of the receive buffer and a new ack, reneging is detected. We implemented the proposed mechanism as a tool called RenegDetect v1.

While verifying RenegDetect v1 with real TCP flows, we discovered that some TCP implementations were generating SACKs incompletely under some circumstances giving a false impression that reneging was happening. Our discovery led us to a side investigation to precisely identify five misbehaving TCP stacks.

For that, we designed a methodology and verified conformant SACK generation on 29 TCP stacks for a wide range of OSes: FreeBSD, Linux, Mac OS X, OpenBSD, Solaris and Windows. We eventually identified the characteristics of seven misbehaviors, and designed seven TBIT tests to document these misbehaviors.

For the first five misbehaviors (A-E) which were observed in the CAIDA trace files, we found at least one misbehaving TCP stack. We reported various versions of OpenBSD and Windows OS to have misbehaving SACK generation implementations. In general, the misbehaving SACK implementations can cause a less efficient SACK-based loss recovery yielding to decreased throughput and longer transfer times.

During the TBIT testing, we identified two additional misbehaviors (F and G). Misbehavior F decreases the throughput by sending less than expected data while using SACKs. Most Linux and OpenSolaris systems show this misbehavior. Misbehavior G is more serious. SACK information from a prior connection reappears in a new connection and can cause a TCP connection to be inconsistent should the sequence number space of one connection overlap that of a prior connection. Solaris

10 and OpenSolaris systems misbehave in this manner. Based on our [RFC2018] SACK generation investigation results, we concluded that while simple in concept, SACK handling is complex to implement.

To identify reneging instances more accurately and identify SACK generation misbehaviors, we updated RenegDetect v2 to better analyze the flow of data, in particular, to analyze data retransmissions which are a more definitive indication that reneging happened.

Our initial hypothesis was that reneging rarely if ever occurs in practice. For that purpose, TCP traces from three domains (Internet backbone (CAIDA), wireless (SIGCOMM), enterprise (LBNL)) were analyzed using RenegDetect v2.

Contrary to our initial expectation that reneging is an extremely rare event, trace analysis demonstrated that reneging does happen. Therefore, we could not reject our initial hypothesis H_0 that P(reneging) < 10^{-5}. Since reneging instances were found, analyzing 300K TCP flows were no longer necessary. As a result, we ended up analyzing 202,877 TCP flows using SACKs from the three domains. In the TCP flows using SACKs, we detected 104 reneging flows. We estimated with 95% confidence that the true average rate of reneging is in the interval [0.041%, 0.059%], roughly 1 flow in 2,000 (0.05%).

In the TCP flows analyzed, we detected 104 reneging flows, or approximately 0.05% in the analyzed TCP connections using SACKs. The frequency of TCP reneging found in [Blanton 2008] was 0.017%. Together the results of these two studies allow us to conclude that reneging is a rare event.

In the 104 reneging flows, a total of 200 reneging instances were detected. This behavior suggests that when reneging occur in a TCP flow, it is much more likely to happen again. It is unclear however if reneging is due to something occurring in the flow, or correlated to what is going on in a host at the given moment in time. For each reneging flow, we tried to fingerprint the operating system of the reneging data receiver, and generalize reneging behavior according to operating system.

Our motivation to investigate the frequency of TCP reneging was primarily to conclude if TCP's design to tolerate reneging is correct. If we could document that reneging never occurs, TCP had no need to tolerate reneging. Upon observing reneging occurs rarely (less than 1 flow per 1000), we believe the current handling of reneging in TCP can be improved.

TCP is designed to tolerate reneging by defining a retransmission policy for a data sender [RFC2018] and keeping the SACKed data in the data sender's send buffer until cumulatively ACKed. With this design, if reneging happens rarely, SACKed data are unnecessarily stored in the send buffer wasting operating system resources.

To understand the potential gains for a protocol that does not tolerate reneging, SCTP's NR-SACKs (Non-Renegable SACKs) are detailed in Section 1.2.2. With NR-SACKs, an SCTP data receiver takes the responsibility for non-renegable data (NR-SACKed), and, an SCTP data sender needs not to retain copies of NR-SACKed data in its send buffer until cumulatively ACKed. Results demonstrated that memory allocated for the send buffer is better utilized with NR-SACKs [Natarajan 2008b]. NR-SACKs also improve end-to-end application throughput. When the send buffer is full, no new data can be transmitted even when congestion and flow control mechanisms allow. When NR-SACKed data are removed from the send buffer, new application data can be read and potentially transmitted. [Yilmaz 2010] shows that the throughput achieved with NR-SACKs is always ≥ throughput observed with SACKs.

If current TCP was designed not to tolerate reneging, the send buffer utilization would be always optimal, and the application throughput might be improved for data transfers with constrained send buffers. Preliminary analysis suggests throughput gains assuming asymmetric buffer sizes (send buffer < receive buffer) and no auto-tuning.

Let us compare TCP's current design to tolerate reneging with a TCP that does not support reneging using the results from our reneging analysis. With current design, TCP tolerates reneging to achieve the reliable data transfers of 104 reneging flows. The 202,773 non-reneging flows waste main memory allocated to send buffer and potentially achieve lower throughput.

I argue that the current design to tolerate reneging is wrong since reneging is a rare event. Instead, I suggest that the current semantics of SACKs should be changed from advisory to permanent prohibiting a data receiver to renege. If a data receiver does have to take back memory that has been allocated to received out-of-order data, I 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 benefit from better send buffer utilization and possible increased throughput.

Initially, reneging was thought as a utility mechanism to help an operating system reclaim main memory under low-memory situations perhaps to avoid a deadlock situation. In our investigation, we found that the average main memory returned to the reneging operating system per reneging instance is on the order of 2 TCP segments (2715, 3717, and 1371 bytes for Linux, FreeBSD, and Windows operating systems, respectively.) This amount of main memory reclaimed seems insignificant. For example, to reclaim 3MB of main memory back to FreeBSD, 846 simultaneous TCP flows each having 3717 bytes of out-of-order data would need to be reneged. On the other hand, our experimentation with FreeBSD showed that terminating a single TCP flow established to Apache web server releases ~3MB of main memory in FreeBSD. I believe that RESETing one TCP flow is a better strategy to help an operating system rather than reneging 800+ connections as would be needed in the current handling of reneging.

I had a chance to discuss why reneging is tolerated in TCP with Matt Mathis, the main editor of [RFC2018]. He told me that the semantics of SACKs are advisory since a reliable data transfer would fail if SACKs were permanent and some TCP stacks implement SACKs incorrectly. By specifying SACKs advisory, TCP is more robust to SACK implementations having bugs. I argue that this design choice which perhaps is practical to get the choice accepted by the research community is wrong. By analogy, a TCP stack implementing a wrong ACK mechanism would cause a data transfer to fail but we do not consider ACKs as advisory. I believe it is the protocol implementor's responsibility to provide a correct implementation. Protocols should be specified to achieve the best performance, and not be designed to tolerate incorrect implementations. I argue that TCP's current mechanism to tolerate reneging achieves a lower memory utilization when compared to a TCP with no reneging support and should be improved.

(2) To investigate the causes reneging, several TCP stacks from popular operating systems were inspected to characterize the circumstances of reneging. The primary contribution of our investigation is that we found out that operating systems use reneging for different purposes.

Initially, reneging was expected when an operating system went low on main memory to help the operating system resume normal operation. FreeBSD, for example, supports that type of reneging. In low memory situations, all TCP connections with out-of-order data renege simultaneously (global reneging).

For Microsoft Windows, I was informed by Dave MacDonald that reneging is not supported by 2000, XP and Server 2003. On the contrary, we found 53 reneging flows in trace analysis where TCP fingerprints strongly suggested these Microsoft systems were reneging. I was also informed by Dave that Vista+ (Vista, Server 2008, 7) comes with a new TCP stack in which reneging is possible. Reneging in Windows Vista+ was introduced to protect a host against DoS attacks. An attacker can open multiple TCP connections and fill each one's receive buffers with out-of-order data to exhaust system resources thus making services unavailable. Reneging happens when the memory consumption of total reassembly data in relation to the global memory limits is significant.

In Mac OS X, reneging is supported by the operating system. But, by default reneging is not enabled. So, TCP can be modified to operate as a non-reneging protocol in Mac OS X.

In Linux (Android), reneging happens when the memory allocated for a receive buffer exceeds the memory limit available to the receive buffer. Allocated buffer space for the out-of-order data is freed and returned back to the global TCP memory pool to be used by other TCP connections. In Linux, only individual connections exceeding the receive buffer limit renege (local reneging).

Reneging is not supported in Solaris but happens to connections where the TCP reassembly queue timer expires (local reneging). To our best knowledge, a timer for the reassembly queue is not defined in the TCP specification. We believe (but could not confirm) reneging in Solaris has the same purpose as Windows reneging: to protect the operating system against a DoS attack.

Initially, we expected reneging not to be supported by any operating systems. To the contrary, our investigation revealed that five out of six inspected operating systems can renege (FreeBSD, Linux (Android), Apple's Mac OS X, Oracle's Solaris and Microsoft's Windows Vista+.) The only operating system that does not support reneging in our investigation is OpenBSD. We also initially expected that reneging would occur to help operating system to resume normal operation by providing extra memory (FreeBSD). Surprisingly, we discovered that reneging is also used as a protection mechanism against DoS attacks (Solaris, Vista+.) We conclude that reneging is a common mechanism implemented in most of today's popular operating systems.

(3) To document the effects of reneging, we designed a tool, CauseReneg, to cause a victim to renege. Using this tool, we attacked various victims deploying popular operating systems with reneging support such as FreeBSD, Linux, Solaris, and Windows using CauseReneg. CauseReneg achieves its goal by exhausting a victim's resources by sending out-of-order data using multiple TCP connections.

For FreeBSD, two attacks were performed to a victim. The first one caused the victim to crash and the second one caused the victim to renege. In both attacks, the available main memory was largely consumed by out-of-order data to trigger reneging. In the first attack, CauseReneg established 1240+ parallel connections to the victim, and the page replacement daemon did not invoke the reneging routines, probably due to low paging activity. In this attack, the victim crashed. Reneging did not help the operating system to resume normal operation. In the second attack, a user process allocating 2GB of main memory was used along with CauseReneg to cause high paging activity and reneging. This time, the page replacement daemon invoked drain routines, and reneging happened. All of the reassembly queues of active TCP connections were purged to reclaim main memory to FreeBSD, and the process using the largest memory allocation was terminated by the page replacement daemon.

Initially, it expected that an operating system starving for main memory would eventually crash. Our first attack to FreeBSD is such an example. In the second attack, when the paging activity was high and the available memory was low, reneging happened in addition to the largest process getting killed. This time, FreeBSD resumed normal operation. The maximum amount of memory that could be allocated to reassembly queues by reneging was limited to ~3MB only (0.6% of the physical memory) for the victim attacked. That amount of memory is negligible compared to the process using the most memory. I believe reneging alone cannot help an operating system to resume normal operation. In this case, the memory used for the reassembly queues was wastefully purged. The operating system resumed normal operation since the page replacement daemon killed the largest process. I argue that reneging support should be turned off by default in FreeBSD as in Mac OS X.

Next, Solaris 11, Windows Vista, and Window 7 victims were attacked using CauseReneg. Both Solaris and Windows use reneging as a protection mechanism against DoS attacks. Solaris uses a uses a reassembly queue timer to renege whereas Windows uses a memory threshold for the out-of-order data for the same purpose.

In Solaris, when out-of-order data sit in the reassembly queue for at least 100 seconds, reneging happens. The system infers that the connection is either terminated due to a failed loss recovery or exhausts resources intentionally (a DoS attack.) In both cases, instead of reneging, terminating the connection with RESETs seems to be a better option. RESETing the connection would release all of the resources held.

In Windows, reneging happens when the memory allocated for out-of-order data exceeds the memory threshold available for the reassembly data. This threshold appears to be ~0.78% of the available physical memory. The current reneging implementation has a potential problem. The out-of-order data that cause reaching the threshold are not reneged. Instead, the out-of-order data received afterwards are reneged. Were an attacker to learn the memory threshold and only send that amount of out-of-order data, all future connections experiencing losses and receiving out-of-order data afterwards would renege. A TCP data sender would not retransmit SACKed data until a retransmission timeout (RTO) [RFC2018]. In such a case, losses would be recovered with RTOs resulting in increased transfer times (lower throughput.) The quality of service, data transfer times for legitimate users, would be reduced. We name this type of an attack a “reduction of service” (RoS) attack. We believe that a RoS attack would be harder to detect compared to a DoS attack since the service provided is not interrupted, but only degraded.

When we compare reneging in Solaris vs. Windows, Solaris’s approach seems to be a better protection mechanism: only the DoS connections are penalized. An important disadvantage of Solaris’s implementation is using a timer since managing a TCP timer is an expensive operation.

Next, we compare global vs. local reneging. FreeBSD employs global reneging. When reneging happens, all the reassembly queues are cleared. On the other hand, Linux and Solaris employ local reneging where each TCP connection reneges independently.

Global reneging is easier to implement. A single reneging function is defined, and no bookkeeping is required. The reneging function is invoked for all active TCP connections when reneging is needed. The disadvantage with global reneging is that if the memory required by the operating system to resume normal operation is less than the total memory allocated for the reassembly queues, some TCP connections are unnecessarily penalized.

Local reneging, on the other hand, is more complex to implement, and requires bookkeeping for each TCP connection and global memory pools. As a connection progresses, the amount of allocated receive buffer space is recorded as data is appended/removed from the receive buffer. In local reneging, only those connections exceeding memory limits experience reneging. Therefore, local reneging is fairer compared to global reneging.

In summary, reneging was caused for FreeBSD, Solaris, and Windows victims using CauseReneg tool. The consequences of reneging were detailed for those systems. When an operating system (e.g. FreeBSD) is starving for memory, reneging alone cannot help the system to resume normal operation. Therefore, I argue that reneging support should be turned off for systems employing that type of reneging. Reneging in Solaris and Windows protects the system against DoS attacks. I argue that type of protection is essential to operating systems but I believe that a better approach would be to RESET the connection under the attack instead of reneging.

In this dissertation, we investigated (1) the instances, (2) causes and (3) effects of TCP reneging in today's Internet to argue that TCP's design to tolerate reneging is wrong. Our investigation showed that reneging is a rare event and in general cannot help an operating system alone to resume normal operation. Therefore, I argue that TCP should be redesigned not to renege by (1) changing semantics of SACKs from being advisory to permanent and (2) RESETing a connection if an operating system has to take back the main memory allocated to out-of-order data, or defend against a DoS attack.

7.2 Future Work

During the reneging analysis on Internet traces, we discovered two additional SACK generation misbehaviors after publishing [Ekiz 2011b]. New TBIT tests are needed to identify the misbehaving stacks for the two additional misbehaviors.

To document the consequences of reneging, FreeBSD, Solaris and Windows hosts were reneged using CauseReneg. Unfortunately, we could not succeed to renege a Linux host yet. In the trace analysis, we detected 40 reneging flows from various Linux data receivers. Deeper investigation showed that reneging happens when the receiving application is unable to read in-order data from the receive buffer. In our attempts to cause reneging on a Linux host, an Apache web server (the receiving application) was reading the in-order data (5 bytes) immediately. To document the consequences of reneging in Linux, CauseReneg needs to be updated to send larger amount of in-order data (10K-20K?) in addition to out-of-order data, and attack an application that would not read in-order data immediately. For that, a custom application that does not read in-order data for long time needs to be developed.

In Section 7.1, we stated that increased throughput for TCP is possible for data transfers with constrained send buffers (assuming asymmetric buffer sizes (send buffer < receive buffer) and no auto-tuning) if TCP were designed not to renege.

When the send buffer is full, no new data can be transmitted even when congestion and flow control mechanisms allow. If SACKs were non-renegable, SACKed data could be removed from the send buffer immediately, and new application data could be read and potentially transmitted if the data receivers receive buffer has space to receive more data. For that, TCP's send buffer management needs to be modified to release SACKed data immediately and read more data into the send buffer with a receipt of a SACK. To document possible throughput improvements, a TCP stack should be modified to operate as a non-renegable TCP and experiments needs to be conducted between two TCP end-points having asymmetric buffer sizes (send buffer < receive buffer) and no auto-tuning.

REFERENCES

[Alexa] ALEXA, "The top 500 sites on the web," [Online]. Available: http://www.alexa.com/topsites.

[Allman 1997] M. Allman, C. Hayes, H. Kruse and S. Ostermann, "TCP performance over satellite links," in *5th International Conference on Telecommunications Systems*, 3/97.

[Apache] The Apache Software Foundation, "Apache HTTP Server Project," [Online]. Available: http://httpd.apache.org/.

[Blanton 2008] J. T. Blanton, "A Study of Transmission Control Protocol Selective Acknowledgement State Lifetime Validity", MS Thesis, November 2008

[Bruning 2005] M. Bruning, "A Comparison of Solaris, Linux, and FreeBSD Kernels," [Online]. Available: http://hub.opensolaris.org/bin/view/Community+Group+advocacy/solaris-linux-freebsd

[Bruyeron 1998] R. Bruyeron, B. Hemon and L. Zhang, "Experimentations with TCP selective acknowledgment," *ACM Computer Communication Review,* vol. 28, no. 2, pp. 54-77, 4/98.

[Caida] CAIDA, "CAIDA Internet Data – Passive Data Sources," [Online]. Available: www.caida.org/data/passive/.

[Dummynet] Dummynet, "Dummynet," [Online]. Available: http://info.iet.unipi.it/~luigi/dummynet/.

[Ekiz 2010] N. Ekiz, P. D. Amer, "A Model for Detecting Transport Layer Data Reneging", PFLDNeT 2010, Lancaster, PA, 11/10

[Ekiz 2011a] N. Ekiz, P. D. Amer, P. Natarajan, R. Stewart and I. Iyengar, "Non-Renegable Selective Acks (NR-SACKs) for SCTP," IETF Internet Draft, draft-natarajan-tsvwg-sctp-nrsack, 08/2011.

[Ekiz 2011b] N. Ekiz, A. H. Rahman and P. D. Amer, "Misbehaviors in TCP SACK Generation," *ACM SIGCOMM Computer Communication Review,* p. 16–23, 2011.

[Ekiz 2011c] N. Ekiz, A. H. Rahman and P. D. Amer, "TBIT tests and TBIT packet captures," [Online]. Available: pel.cis.udel.edu/tbit-tests/, 2/11

[Fall 1996] K. Fall and S. Floyd, "Simulation-based comparisons of Tahoe, Reno, and SACK TCP," *ACM Computer Communication Review,* vol. 26, no. 3, pp. 5-21, 6/96.

[Fisk 2001] M. Fisk, W. Feng, "Dynamic Right-Sizing: TCP Flow-Control Adaptation", Proc. of ACM/IEEE Supercomputing Conference, pp. 1-3, 11/01

[Fraleigh 2003] C. Fraleigh, S. Moon, B. Lyles, C. Cotton, M. Khan, D. Moll, R. Rockell, T. Seely and C. Diot, "Packet-Level Traffic Measurements from the Sprint IP Backbone," *IEEE Network,* vol. 17, no. 6, pp. 6-16, 11/03.

[Freebsd] The FreeBSD Foundation, "FreeBSD," [Online]. Available: www.freebsd.org.

[FreebsdImpl] FreeBSD TCP and SCTP Implementation, October 2007. [Online]. Available: www.freebsd.org/cgi/cvsweb.cgi/src/sys/netinet

[Iyengar 2006] J. R. Iyengar, P. D. Amer and R. Stewart, "Concurrent Multipath Transfer using SCTP Multihoming Over Independent End-to-End Paths," *IEEE/ACM Transactions on Networking,* vol. 14, no. 5, pp. 951-964, 10/06.

[Iyengar 2007] J. R. Iyengar, P. D. Amer and R. Stewart, "Performance implications of a bounded receive buffer in concurrent multipath transfer," *Computer Communications,* vol. 30, no. 4, pp. 818-829, 2/07.

[Jaiswal 2004] S. Jaiswal, G. Iannaccone, C. Diot, J. Kurose and D. Towsley, "Inferring TCP Connection Characteristics Through Passive Measurements," in *Proc. IEEE INFOCOMM*, 3/04.

[Ladha 2004] S. Ladha, P. D. Amer, A. J. Caro and J. R. Iyengar, "On the Prevalence and Evaluation of Recent TCP Enhancements," in *IEEE Globecom*, 11/04.

[LBNL 2004] Lawrence Berkeley National Laboratory, "LBNL/ISCI Enterprise Tracing Project," [Online]. Available: http://www.icir.org/enterprise-tracing/

[Linux] Linux Kernel Organization, Inc., "The Linux Kernel Archives," [Online]. Available: http://www.kernel.org/.

[MacOS] Mac OS TCP Implementation, [Online]. Available: http://fxr.watson.org/fxr/source/bsd/netinet/?v=xnu-1699.24.8;im=3.

[Market] Net Applications, "Operating System Market Share," [Online]. Available: http://marketshare.hitslink.com/operating-system-market-share.aspx?qprid=8.

[MacDonald 2000] D. MacDonald and W. Barkley, "Microsoft Windows 2000 TCP/IP Implementation Details," Microsoft, [Online]. Available: http://technet.microsoft.com/en-us/library/bb726981.aspx.

[Medina 2004] A. Medina, M. Allman and S. Floyd, "Measuring interactions between transport protocols and middleboxes," in *Proc. of ACM SIGCOMM*, 10/04.

[Medina 2005] A. Medina, M. Allman and S. Floyd, "Measuring the Evolution of Transport Protocols in the Internet," *ACM SIGCOMM Computer Communication Review,* vol. 35, no. 2, pp. 52-66, 4/05.

[Moore 1993] D. S. Moore, G. P. McCabe, "Introduction to the Practice of Statistics", Freeman, New York, 1993

[Natarajan 2006] P. Natarajan, J. R. Iyengar, P. D. Amer and R. Stewart, "Concurrent Multipath Transfer Using Transport Layer Multihoming: Performance during Network Failures," in *MILCOM 2006*, 10/06.

[Natarajan 2008a] P. Natarajan, N. Ekiz, P. D. Amer, J. R. Iyengar and R. Stewart, "Concurrent Multipath Transfer using SCTP Multihoming: Introducing the Potentially-Failed Destination State," in *IFIP Networking 2008*, 5/08.

[Natarajan 2008b] P. Natarajan, N. Ekiz, E. Yilmaz, P. D. Amer, J. R. Iyengar and R. Stewart, "Non-renegable selective acks (NR-SACKs) for SCTP," in *Int'l Conf on Network Protocols (ICNP)*, 10/08.

[Natarajan 2009] P. Natarajan, N. Ekiz, P. D. Amer and R. Stewart, "Concurrent Multipath Transfer during path failure," *Computer Communications,* vol. 32, no. 15, pp. 1577-1587, 9/09.

[Nmap] Nmap, "nmap," [Online]. Available: www.nmap.org.

[Ns-2] "The Network Simulator - ns-2," [Online]. Available: http://www.isi.edu/nsnam/ns/.

[Openbsd] OpenBSD Foundation, "OpenBSD," [Online]. Available: http://www.openbsd.org.

[Padhye 2001] J. Padhye and S. Floyd, "On inferring TCP behavior," in *ACM SIGCOMM*, 8/01.

[Paxson 1997] V. Paxson, "Automated Packet Trace Analysis of TCP Implementations," *ACM SIGCOMM Computer Communication Review,* vol. 27, no. 4, pp. 167-179, 9/97.

[RFC793] J. Postel, "Transmission Control Protocol," RFC 793, 10/81.

[RFC1122] R. Braden, "Requirements for Internet Hosts -- Communication Layers," RFC 1122, 10/89

[RFC1323] V. Jacobson, R. Braden, D. Borman, "TCP Extensions for High Performance," RFC 1323, 05/92

[RFC1945] T. Berners-Lee, R. T. Fielding and H. F. Nielsen, "Hypertext Transfer Protocol -- HTTP/1.0," RFC 1945, 5/96.

[RFC2018] M. Mathis, J. Mahdavi, S. Floyd and A. Romanow, "TCP Selective Acknowledgment Options," RFC 2018, 9/96.

[RFC2119] S. Bradner, "Key words to use in RFCs to Indicate Requirement Levels," RFC 2119, 3/97

[RFC2616] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach and T. Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1," RFC 2616, 6/99.

[RFC2883] S. Floyd, J. Mahdavi, M. Mathis and P. Matthew, "An Extension to the Selective Acknowledgement (SACK) Option for TCP," RFC 2883, 7/00.

[RFC3390] M. Allman, S. Floyd, C. Partridge, "Increasing TCP's Initial Window," RFC 3390, 10/02

[RFC3517] E. Blanton, M. Allman, K. Fall and L. Wang, "A Conservative Selective Acknowledgment (SACK)-based Loss Recovery Algorithm for TCP," RFC 3517, 4/03.

[RFC4960] R. Stewart, "Stream Control Transmission Protocol," RFC 4960, 9/07.

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

[Sigcomm 2008] SIGCOMM 2008 Traces, [Online]. Available: http://www.cs.umd.edu/projects/wifidelity/sigcomm08_traces/

[Singh 2003] A. Singh, "What is Mac OS X?," [Online]. Available: http://osxbook.com/book/bonus/ancient/whatismacosx/arch_xnu.html, 12/03

[Tbit] TBIT, "The TCP Behavior Inference Tool," [Online]. Available: www.icir.org/tbit/.

[Tcpdump] Tcpdump, "Tcpdump," [Online]. Available: www.tcpdump.org.

[Virtualbox] VirtualBox, "VirtualBox," [Online]. Available: www.virtualbox.org.

[Windows 2003] Microsoft Corporation, "Microsoft Windows Server 2003 TCP/IP Implementation Details", 06/2003

[Wireshark] Wireshark, "Wireshark," [Online]. Available: www.wireshark.org.

[Yilmaz 2010] E. Yilmaz, N. Ekiz, P. Natarajan, P. D. Amer, J. T. Leighton, F. Baker and R. R. Stewart, "Throughput analysis of Non-Renegable Selective Acknowledgments (NR-SACKs) for SCTP," *Computer Communications,* vol. 33, no. 16, pp. 1982-1991 , 10/10.