A Model for Detecting Transport Layer Data Reneging

Nasif Ekiz
Computer and Information Sciences Department
University of Delaware
Newark, Delaware 19716
nekiz@udel.edu

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

ABSTRACT

Data reneging occurs when a data receiver first SACKs data, and later discards that data from its receiver buffer prior to delivering it to the receiving application or socket buffer. Today's reliable transport protocols such as TCP and SCTP are designed to tolerate data reneging. We argue that this design assumption is wrong, in part based on a hypothesis that data reneging rarely if ever occurs in practice. To support our hypothesis, we present a model for detecting instances of data reneging by analyzing traces of TCP traffic. Using this model, we will investigate the frequency of data reneging in Internet traces provided by CAIDA.

Categories and Subject Descriptors

C.2.5 [**Local and Wide-Area Networks**]: *Internet (e.g., TCP),* C.2.6 **[Internetworking]**: *Standards (TCP, SACK)*

General Terms

Measurement, Verification.

Keywords

Data reneging, SACK, SCTP, TCP.

1. INTRODUCTION

Transmission Control Protocol (TCP) [14] uses sequence numbers and cumulative acknowledgments (ACKs) to achieve reliable data transfer. A TCP data receiver uses sequence numbers to sort arrived data segments. Data arriving in expected order, i.e., *ordered data*, is cumulatively ACKed (herein ACKed) to the data sender. The data sender assumes the data receiver accepts responsibility of delivering ACKed data to the receiving application, and deletes all ACKed data from its send buffer, potentially even before that data is delivered to a receiving application.

The 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. A correct TCP data receiver implementation must not delete ACKed data without first delivering it to the receiving application since the data sender may remove ACKed data from its send buffer.

The Selective Acknowledgment Option (SACK), specified in RFC 2018 [9], is an extension to TCP's cumulative ACK mechanism, and is used by a data receiver to acknowledge (herein SACK) arrived out-of-order data to the data sender. The intent is that SACKed data do not need to be retransmitted during loss recovery. Prior research [1, 2, 5] showed that SACK improves TCP throughput when multiple losses occur in the same window.

Deployment of the SACK option in TCP connections is an increasing trend. In 2001, 41% of the web servers tested were SACK-enabled [12]. In 2005, SACK-enabled web servers increased to 68% [10]. All recent versions of FreeBSD, Linux, Mac OS, OpenBSD, OpenSolaris, Solaris, and Windows create SACK-enabled TCP connections by default.

Data receiver reneging (herein *data 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 or socket buffer. TCP is designed to tolerate data reneging. Specifically RFC 2018 states: "*The SACK option is advisory, in that, while it notifies the data sender that the data receiver has received the indicated segments, the data receiver is permitted to later discard data which have been reported in a SACK option*". Data reneging might happen, for example, when an operating system needs to recapture previously allocated memory for another process, say to avoid deadlock. Data reneging might happen in operating systems such as FreeBSD, Linux and Mac OS. For example, in FreeBSD and Mac OS, the *sysctl* option *net.inet.tcp.do_tcpdrain* turns on/off data reneging support [7].

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

We argue that SACK's design assumption to tolerate data reneging is wrong. This opinion is based on: (1) a hypothesis that data reneging rarely if ever occurs in practice, and (2) research demonstrating potential improved performance if SACKed data were not renegable.

In Section 2, we further present the motivation to detect data reneging instances. Then Section 3 presents the model to detect data reneging instances based on Internet trace files provided by Cooperative Association for Internet Data Analysis (CAIDA) [15]. Section 4 presents results of verifying our model. Section 5 identifies several past methodologies to infer TCP behavior, and

Section 6 presents our on-going research to apply the model to TCP traces.

2. DOES DATA RENEGING HAPPEN?

Data reneging is a transport layer behavior of which we know little about its frequency of occurrence in practice. This section provides motivation to detect data reneging instances in reliable transport protocols such as TCP and SCTP.

To motivate the study of data reneging, we first need to understand the potential gains of a transport protocol that does not tolerate data reneging. For that, we present a brief background on Non-Renegable Selective Acks (NR-SACKs) [4].

2.1 NR-SACKs

NR-SACK is a new ack mechanism proposed for the Stream Control Transmission Protocol (SCTP) [16]. With the NR-SACK extension, an SCTP data receiver takes responsibility for selectively acked data (NR-SACKed). In that case, an SCTP data sender no longer needs to retain copies of NR-SACKed data in its send buffer until ACKed. Just as with ACKed data, NR-SACKed data can be removed from the send buffer immediately on the receipt of the NR-SACK.

With NR-SACKs, the main memory allocated for the send buffer is better utilized. Natarajan et al. [11] present send buffer utilization results for unordered data transfers over SCTP under mild (~1-2%), medium (~3-4%) and heavy (~8-9%) loss rates for NR-SACKs vs. SACKs. For the bandwidth-delay parameters studied, the memory wasted by assuming SACKed data could be reneged was on average ~10%, ~20% and ~30% for the given loss rates, respectively.

NR-SACKs also can 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 is removed from the send buffer, new application data can be read and potentially transmitted.

Yilmaz et al. [17] investigate throughput improvements for NR-SACK vs. SACK. The authors show that the throughput achieved with NR-SACKs is always ≥ the 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.

2.2 Motivation to Study Data Reneging

Consider designing reliable transport protocols to NOT tolerate data reneging. In such a case, the send buffer utilization would be always optimal, and the application throughput could be improved for data transfers with constrained send buffers. Current transport protocols employing SACKs (TCP, SCTP) suffer because of the assumption that data reneging may happen.

If we can document that data reneging never happens or happens rarely, we can argue that reliable transport protocols should be modified to assume all selectively acked data is non-renegable. As a simple example, assume that data reneging happens rarely, say once in 100,000 TCP flows.

Case A (current practice): TCP tolerates data reneging to achieve reliable data transfer in a single data reneging connection. 99,999 non-reneging connections *potentially* waste memory allocated for send buffer, and achieve lower application throughput. One reneging connection operates without interruption.

Changing transport protocols that currently support data reneging into non-reneging transport protocols requires minor modification. First, the semantics for SACK are changed from *advisory* to *permanent*. Second, if a data receiver does have to renege, we propose the data receiver must RESET the connection.

Case B (proposed change): TCP does not tolerate data reneging. 99,999 non-reneging connections potentially have improved performance, and 1 reneging connection is aborted. (Given the dire situations requiring a receiver to renege, aborting the reneging connection is unlikely to make matters worse.)

We hypothesize that few (if any) connections will be penalized, and the large majority of non-reneging connections will potentially benefit from better send buffer utilization and increased throughput. The problem is that data reneging has never been studied by the research community. No one knows what percentage of connections renege. The key issue – does data reneging occur or not?

3. A MODEL TO DETECT RENEGING

To begin to answer this key issue, this section presents a model to passively detect data reneging instances occurring in Internet traces. First, we present how a TCP or SCTP data sender infers data reneging in sections 3.1 and 3.2, respectively. In section 3.3, we introduce our model to detect data reneging instances.

3.1 Detecting Reneging at TCP Data Sender

In the current TCP and TCP SACK specifications, a TCP data sender has no design to infer data reneging. To tolerate data reneging, a TCP data sender keeps copies of SACKed data in its send buffer until that data is ACKed. To achieve reliable data transfer, the following retransmission policy is specified in [9] for a data sender 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 timeout happens. At the timeout, the TCP data sender clears all “SACKed” information due to possible data reneging, and retransmits the segment at the left edge of the send buffer.

3.2 Detecting Reneging at SCTP Data Sender

SCTP supports data reneging detection at the data sender. Unlike TCP’s constrained number on the reported SACK options (4 at maximum), an SCTP data receiver can generate SACK chunks with a large number of SACK options. For example, for a path with MTU=512 bytes, a SACK chunk can report 116 SACK options (20 bytes for IP header, 12 bytes for SCTP common header, 16 bytes for SACK chunk header + 116 * 4 byte SACK options).

Thus, an SCTP data sender receives a more accurate view of the data receiver’s buffer, and can accurately infer data reneging by inspecting SACK options. If a new SACK arrives and previously

SACKed data is not present, the SCTP data sender infers data reneging, and marks the reneged data for retransmission.

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

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

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

ACK 2 SACK 7-7 is sent when data 7 arrives out-of-order. This SACK also implies data reneging (for data 3-6) if the previous ACK 2 was lost.

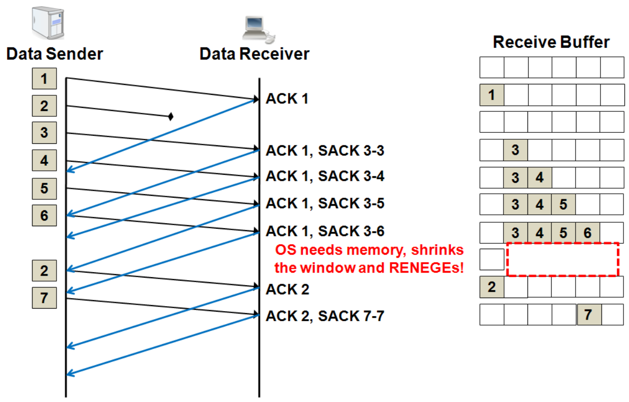

Figure 1. Detecting data reneging at SCTP data sender

3.3 Detecting Reneging at a Router

To detect an SCTP data reneging instance, a data sender infers the state of the data receiver's receive buffer through ACKs and SACKs. Even though TCP has no mechanism to detect data reneging instances, data reneging can be detected by analyzing TCP ack traffic, and inferring the state of the receiver's buffer.

For a TCP data receiver, the state of the receive buffer can be learned with the ACKs and SACKs, and updated through the new acks observed at an intermediate router. The state consists of a cumulative ACK value (stateACK) and a list of out-of-order data blocks (stateSACK blocks) known to be in the receive buffer.

The example in Figure 1 assumed all ack traffic arrives to the data sender and data reneging is detected. Consider the example scenario when the ack traffic is monitored by an intermediate router. In the example, the data 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 2 shows the same data transfer where only three acks are monitored at the intermediate router. On seeing ACK 1 SACK 3-4, the router determines the state of receive buffer is: ordered data 1 is delivered or deliverable to the receiving application (stateACK 1) and out-of-order data 3-4 is in the receive buffer (stateSACK 3-4). ACK 1 SACK 3-6 updates this state by adding out-of-order data 5-6 as SACKed (stateSACK 3-6). When ACK 2 SACK 7-7 is received and compared to the state of receive buffer (stateACK 1, stateSACK 3-6), an inconsistency is observed and data reneging is detected since data 3-6 are not SACKed.

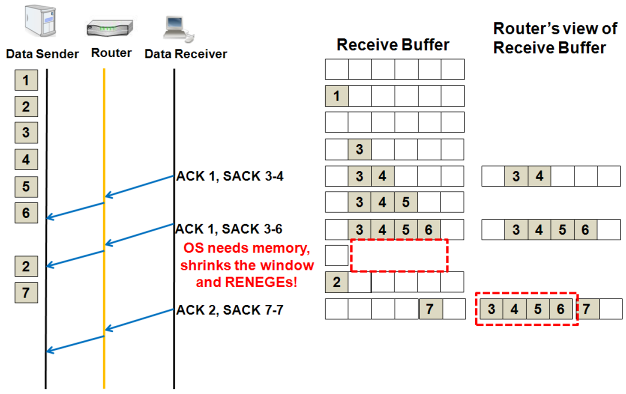

Figure 2. Detecting data reneging by an intermediate router

Even though the number of acks observed at the intermediate router was limited, the state of the receive buffer is as for Figure 1. Because a SACK option reports all consecutive out-of-order segments as a block, the intermediate router can infer the complete state of the receive buffer most of the time.

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 at a data receiver is more than four, then 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 1 presents the number of SACK options 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 options can be included in a TCP segment. For segments with 1, 2, or 3 SACK option(s), the TCP header length is checked to determine if another SACK option could have been appended to the TCP header. TCP segments with 4 SACK options 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 would be still enough to detect data reneging instances. The reasoning is that we expect a reneging data receiver will purge all of the out-of-order data as occurs in FreeBSD [7]. Since the intermediate router has state information about out-of-order data, data reneging instance will be detected when acks with no SACK option are observed.

Table 1. Number of SACK options in TCP segments

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

Our software to detect data reneging instances (*Reneg-detect*) constructs the state of the receive buffer for TCP flows 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 receive buffer state is updated. Otherwise data reneging instance is detected and reported.

We now describe our model for constructing the state of receive buffer at an intermediate router. The state consists of a cumulative ACK value and a list of ordered out-of-order data blocks (SACK blocks) known to be in the receive buffer.

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

Figure 3 presents our model for constructing and updating the SACK block state of the receive buffer. The state is initialized with the first TCP ack observed in a flow. If the ack has no SACK option(s), only the cumulative ACK value is recorded. If the ack includes SACK option(s), each one is added as a SACK block to the state.

When the next TCP ack is observed, each reported SACK option (corresponding to a New SACK Block (N) in Figure 3) is compared with the SACK blocks in the receive buffer state. Each SACK block in the receive buffer state is represented by Current SACK Block (C) in Figure 3.

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 ($(N \cap C \neq \emptyset) \wedge !(N \supseteq C) \wedge !(N \supset C)$)
4. N does not intersect with C ($N \cap C = \emptyset$)

Note that the above cases are mutually exclusive. Each case is described in turn. For the given examples, assume an initial receive buffer state as follows: the cumulative ACK value is 8 (stateACK 8), and there is one SACK Block (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 is updated to reflect the new SACK block. The update may be in terms of a left edge extension, a right edge extension or both. After the update, the new SACK block is compared with the next SACK blocks in the state. The reasoning is that a new SACK block may be the superset of a number of SACK blocks in the receive buffer state due to possible ack reordering, and may fill a gap between two SACK blocks.

Case 2: When a new SACK block (e.g., SACK 12-13) is a proper subset of a current SACK block (e.g., stateSACK 12-15), the comparison implies data reneging (out-of-order data 14-15 had been deleted from the receive buffer). An instance of data reneging is logged for future deeper analysis.

Case 3: Data reneging is detected similarly when a new SACK block (e.g., SACK 14-20) intersects with a current SACK block (stateSACK 12-15). 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, 16-20, as well as the removal of previously SACKed data, 12-13. The state is not updated (to catch more inconsistencies) until the cumulative ACK is advanced beyond the SACK blocks that trigger data reneging instances.

Case 4: If a new SACK block (e.g., SACK 22-25) and a current SACK block (e.g., stateSACK 12-15) do not intersect, the new SACK block is compared with the next SACK block in the state. If the new SACK block reported is disjoint with all of the SACK blocks in the state, the new SACK block is added to the receive buffer state. The updated receive buffer state becomes stateACK 8, $stateSACK_1$ 12-15, $stateSACK_2$ 22-25.

The model detects data reneging instances only when some SACK options are included in the acks. If a data receiver purges all out-of-order data in the receive buffer, no SACK options are reported. In such a case, the receive buffer state would have a number of SACK blocks, and the new ack reports no SACK blocks (even though TCP options field has enough space to report at least one SACK option). *Reneg-detect* also infers such data reneging instances.

Data reneging may be inferred spuriously if acks are reordered before arriving at the intermediate router. To cope with 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.

4. MODEL VERIFICATION

Reneg-detect was verified with synthetic TCP flows that mimic data reneging behavior. Data reneging flows were created using *text2pcap* tool, and all of the data reneging flows tested were identified correctly as reneging.

Reneg-detect also was verified by analyzing 100s of TCP flows from Internet traces provided by CAIDA. Initially it seemed that data reneging was happening frequently. On closer inspection however, it turned out that the generation of SACKs in many TCP implementations was incorrect (!) according to RFC 2018. Sometimes SACK information that should have been sent was not. Sometimes the wrong SACK information was sent. These misbehaviors wrongly gave the impression that data reneging was occurring.

Our discovery led us to a side investigation to verify SACK generation behavior of TCP data receivers for a wide range of operating systems [3]. Now, we are developing a methodology for verifying SACK behavior, and we will apply the methodology to report misbehaving TCP stacks.

Based on the results of the model verification effort, we updated *Reneg-detect* to identify these misbehaviors, and not report them as instances of data reneging.

5. RELATED WORK

Previous studies employed passive measurements to infer specific protocol behavior by analyzing large number of TCP flows. In those passive measurement studies, collected trace files were analyzed to infer the specific TCP behavior.

Paxson [13] presents *tcpanaly,* a tool which automatically analyses the correctness of TCP implementations by inspecting traces collected for bulk data transfers.

Fraleigh [6] 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 and 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.

In Jaiswal [8], the authors introduce a passive measurement technique to infer and keep track of congestion window (cwnd) and round trip time (RTT) of a TCP data sender. To infer data senders' cwnd, the authors construct a replica of the data sender's TCP state using a finite state machine (FSM). FSM is updated through ACKs and retransmissions seen at the data collection point.

6. WORK IN PROGRESS

To detect data reneging instances, we need TCP flows in which some SACK options are observed during the data transfer. For that, we are filtering CAIDA traces to obtain only TCP flows with SACK options to analyze them with *Reneg-detect*.

The summary of Internet trace files provided by CAIDA by (year/data collection machine/number of traces available) is as follows:

- 2008/equinix-chicago/10
- 2008/equinix-sanjose/6
- 2009/equinix-chicago/12
- 2009/equinix-sanjose/12
- 2010/equinix-chicago/3
- 2010/equinix-sanjose/3

The total duration of each trace is 1 hour and consists of 60 one minute traces. In our lab we do not have enough computation power to analyze all of the traces provided. Instead we are planning to analyze TCP flows from each data set with total duration of 2-3 minutes. The minutes to be used will be chosen randomly.

We are also looking for TCP trace files from other domains such as wireless networks where the loss rate is higher. Our goal is to analyze millions of TCP flows using *Reneg-detect*, and document the frequency of data reneging instances. Based on these empirical observations, we will provide the first documentation of transport layer data reneging in the literature.

7. ACKNOWLEDGMENTS

The authors would like to thank Abuthahir Habeeb Rahman, Jonathan Leighton, Aasheesh Kolli and Ersin Ozkan for the valuable discussions and comments while developing this paper. This work is supported by the University Research Program, Cisco Systems, Inc.

8. REFERENCES

[1] Allman, M., Hayes, C., Kruse H., and Ostermann, S. 1997. TCP performance over satellite links. *5th Int'l Conf. on Telecommunications Systems*. 3 (1997).

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

[3] Ekiz, N., Rahman, A. H., and Amer, P. D. Misbehaviors in SACK generation (submitted for publication).

[4] Ekiz, N., Amer, P., Natarajan, P., Stewart, R., and Iyengar, J. 2010. Non-renegable selective acks (NR-SACKs) for SCTP. IETF Internet Draft (work in progress). tools.ietf.org/id/draft-natarajan-tsvwg-sctp-nrsack-06.txt

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

[6] Fraleigh, C., Moon, S., Lyles, B., Cotton, C., Khan, M., Moll, D., Rockell, R., Seely, T., and Diot, S. C. 2003. Packet-Level traffic measurements from the Sprint IP backbone. *IEEE Network*. 17, 6 (Nov. 2003), 6-16. DOI= http://dx.doi.org/10.1109/MNET.2003.1248656.

[7] FreeBSD TCP Implementation. www.freebsd.org/cgi/cvsweb.cgi/src/sys/netinet/

[8] Jaiswal, S., Iannaccone, G., Diot, S. C., Kurose, J., and Towsley, D. 2004. Inferring TCP connection characteristics through passive measurements. *IEEE INFOCOM*, 3 (Mar. 2004), 1582-1592. DOI= http://dx.doi.org/10.1109/INFCOM.2004.1354571.

[9] Mathis, M., Mahdavi, J., Floyd, S., and Romanow, A. 1996. TCP selective acknowledgment options. RFC 2018.

[10] Medina, A., Allman, M., and Floyd, S. 2005. Measuring the evolution of transport protocols in the internet. *SIGCOMM Comput. Commun. Rev.* 35, 2 (Apr. 2005), 37-52. DOI= http://doi.acm.org/10.1145/1064413.1064418.

[11] Natarajan, P., Ekiz, N., Yilmaz, E., Amer, P. D., Iyengar, J., and Stewart, R. 2008. Non-Renegable Selective Acknowledgments (NR-SACKs) for SCTP. *IEEE International Conference on Network Protocols*. (October 2008), 187-196. DOI= http://dx.doi.org/10.1109/ICNP.2008.4697037.

[12] Pahdye, J. and Floyd, S. 2001. On inferring TCP behavior. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols For Computer Communications* (San Diego, California). SIGCOMM '01. ACM, New York, NY, 287-298. DOI= http://doi.acm.org/10.1145/383059.383083.

[13] Paxson, V. 1997. Automated packet trace analysis of TCP implementations. In *Proceedings of the ACM SIGCOMM '97 Conference on Applications, Technologies, Architectures, and Protocols For Computer Communication* (Cannes, France, Sep. 14 - 18, 1997). M. Steenstrup, Ed. SIGCOMM '97. ACM, New York, NY, 167-179. DOI= http://doi.acm.org/10.1145/263105.263160.

[14] Postel, J. 1981. Transmission control protocol. RFC 793.

[15] Shannon, C., Aben E., Claffy, K., Andersen, D., The CAIDA Anonymized 2008, 2009, 2010 Internet Traces. http://www.caida.org/data/passive/

[16] Stewart, R. 2007. Stream control transmission protocol. RFC 4960.

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

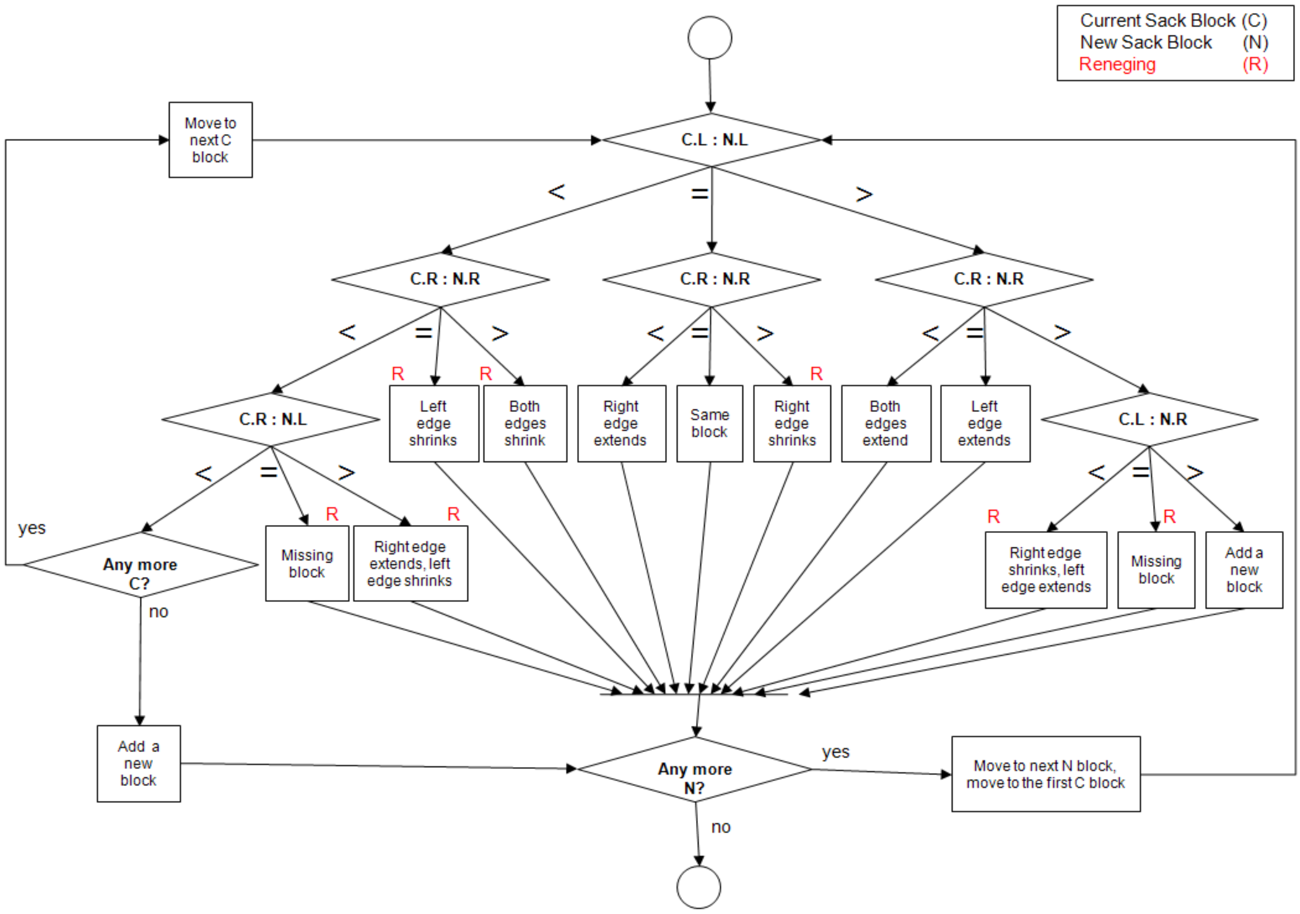

Figure 3. Data Reneging Detection Model