An Implementation of the SACK-Based Conservative Loss Recovery Algorithm for TCP in ns-3 (extended version)

Truc Anh N. Nguyen* and James P.G. Sterbenz*†§
*Information and Telecommunication Technology Center
Department of Electrical Engineering and Computer Science
The University of Kansas, Lawrence, KS 66045, USA
†School of Computing and Communications (SCC) and InfoLab21
Lancaster University, LA1 4WA, UK
§Department of Computing
The Hong Kong Polytechnic University, Hung Hom, Kowloon, Hong Kong
{annguyen, jpgs}@ittc.ku.edu
www.ittc.ku.edu/resilinets

ABSTRACT

This paper presents an implementation of the SACK option and TCP SACK in ns-3. The paper then discusses the verification of the implementation and analyzes the performance of TCP SACK under different network scenarios, especially those with high loss rate and with the presence of correlated losses. The paper also compares TCP SACK's performance with other TCP variants: Tahoe, Reno, NewReno, and Westwood+.

Categories and Subject Descriptors

I.6 [**Simulation and Modeling**]: General, Model Development, Model Validation and Analysis

General Terms

Implementation, Analysis, Testing, Verification

Keywords

TCP SACK, Tahoe, Reno, NewReno, Westwood+, ns-3 simulation

1. INTRODUCTION

TCP is the dominant reliable transport protocol for the Internet and has evolved over time with extensive studies, modifications, and extensions to improve its performance in various network environments. Being developed for wired networks, the traditional TCP fails to maintain its throughput when being deployed in other environments partly due to its inability to handle corruption-based losses in an efficient way [22]. With the increasing deployment of wireless networks, satellite links, or mobile Internet in which not only loss rate is high, but highly correlated losses are found to be common, TCP loss recovery has become an interesting research topic. Over the years, many mechanisms have been developed to improve the loss recovery process. One of those mechanisms is the SACK (selective ACK) option extension to TCP [25], which in turn motivates the development of TCP SACK's conservative loss recovery algorithm [5, 6]. The details of these developments are explained in the next section.

Given the lack of the SACK option and TCP SACK models in ns-3 [1], we have decided to implement these models to extend ns-3 functionality and hopefully enable further developments and studies that are based on or make use of our models. After the implementation, we verify our models with regression tests to ensure they are error-free and compatible with the other existing models in the ns-3 simulator. Finally, we simulate TCP SACK under correlated losses to study its behavior and compare performance with existing TCP variants in ns-3 including TCP Tahoe [17], Reno [18, 4], NewReno [16, 10], and Westwood+ [24].

The remainder of this paper is organized as follows: Section 2 presents a brief discussion on the SACK option and TCP SACK followed by a short survey of other related work. Section 3 describes the implementation of the SACK option and TCP SACK in ns-3, followed by a brief discussion on the verification method. Section 4 presents the simulation setup and analysis to study the behavior of TCP SACK. Finally, Section 5 concludes the paper with directions for future work.

2. BACKGROUND AND RELATED WORK

The SACK option is a TCP extension that addresses the limitation of traditional cumulative go-back-N ACKing in facing multiple segment losses per sending window. The option was first proposed in RFC 1072 [19] and revised in RFC 2018 [25] to increase its robustness. This option allows a TCP receiver to inform the data sender those segments that have been received but cannot be acknowledged by the cumulative ACK due to the existence of a gap in the receiver's buffer. With this SACK information, the sender can avoid duplicate retransmissions and can recover multiple losses in a sending window to speed up the recovery process.

The SACK extension consists of 2 options: the SACK-

Extended version of Wns3 2015 May 13-14, Barcelona, Spain.

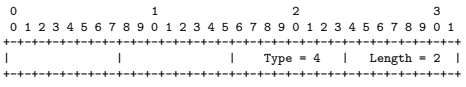

Figure 1: Sack-Permitted Option [19, 25]

PERMITTED and the SACK option. SACK-PERMITTED is appended to the SYN message to negotiate the use of SACK throughout a connection lifetime. This is a simple 2-byte option with a type of 4 as illustrated in Figure 1. The SACK option depicted in Figure 2 contains multiple SACK blocks in which each block represents a contiguous but isolated chunk of data in the receiver's buffer. Each block is defined by two 32-bit byte-sequence numbers called left edge and right edge marking the beginning and the end of the block, respectively. Although the receiving host is supposed to report as many blocks as needed, the 40-byte limitation on TCP options imposes a constraint on the maximum number of blocks allowed to 3 blocks, given the typical presence of the TCP timestamp option [20]. In addition, the order of all SACK blocks within a SACK option is crucial for this TCP extension to achieve its full potential. The first block should cover the most recently received octets while the remaining blocks should be constructed with the most recently reported blocks appearing first.

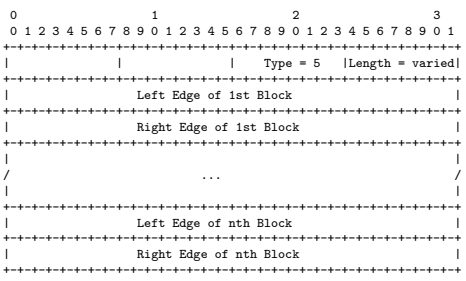

Figure 2: SACK Option [25]

While being steadily deployed in the Internet, the SACK option is also an interesting research topic with extensive studies. SACK has been extended to D-SACK (Duplicate-SACK) to allow the reporting of duplicate segments caused by network's replication, packet reordering, ACK losses, or a premature RTO [11]. D-SACK is employed in RR-TCP (Reordering-Robust TCP) to improve TCP throughput in networks suffering significant packet reordering [30]. ISACK (Improved SACK) [21] is another SACK modification to improve bit efficiency, which has been known as one of the drawbacks of SACK, particularly in bandwidth-constrained environments. The development of ISACK leads to the proposal of a new ASACK (Adaptive SACK) strategy to dynamically switch between ISACK and SACK [21]. NR-SACK (Non-Renegable SACK) [26] modifies SACK's behavior to inform which out-of-order data the receiver will not discard. This allows the sender to free up its buffer space before receiving cumulative ACKs. When employed in SCTP (Stream Control Transmission Protocol) [28], the use of NR-SACK results in better utilization of the sending buffer's memory [26]. NR-SACK helps improve MPTCP (Multipath TCP) [12] overall throughput in the case where the sending buffer size becomes the bottleneck in the presence of multiple TCP flows [29].

TCP SACK was proposed in RFC 3517 [5] and refined in RFC 6675 [6], and implements a conservative loss recovery algorithm that exploits the information carried in SACK option to assist the sender in making retransmission decision and better utilizing the available network bandwidth. In addition to NewReno's fast recovery, this algorithm is another loss recovery scheme developed to enhance TCP performance in the presence of multiple losses. A simulation study when comparing between Tahoe, Reno, and TCP SACK showed that the employment of SACK allows TCP to recover faster from losses and to avoid duplicate retransmissions [9]. When TCP with the SACK extension is used in a satellite environment, the performance can be improved in some cases in which congestion and high bit error rates are present [3]. A study on the effect of different TCP enhancements including SACK and TCP SACK on the overall performance showed that while SACK outperforms NewReno when the loss rate exceeds 9% for bulk transfers, it does not offer noticeable improvements for Web transfers [23]. When TCP SACK performance is analyzed under highly correlated losses, it does not offer any performance gain when compared with TCP Reno [15]. There are two reasons for this result: first, the SACK recovery time might take longer than a timeout; second, the probability of a successful SACK recovery is not higher than the probability of a timeout occurrence [15]. However, both of the studies [23, 15] implement the original TCP SACK specification, RFC 3517 [5]. To the best of our knowledge, there have not been any studies that implement the recent changes specified in RFC 6675 [6].

3. TCP SACK MODULE FOR ns-3

TCP SACK is implemented as a derived class of the `TcpSocketBase` in the Internet module that houses other TCP variants including the early TCP implementation with no congestion control (`TcpRfc793`), TCP Tahoe (`TcpTahoe`), TCP Reno (`TcpReno`), TCP NewReno (`TcpNewReno`), and TCP Westwood(+) (`TcpWestwood`). The ns-3 models of TCP Westwood and Westwood+ were also implemented by the ResiliNets Research Group [14]. In this section, we explain the implementation details, starting with an overview of the TCP module in ns-3, followed by a discussion of the SACK and SACK-PERMITTED option implementation. The section is followed with a list of important global variables a SACK sender holds to make appropriate loss recovery action upon the arrival of SACK information from the other side. The TCP SACK important feature, the *scoreboard* implementation is also presented. The section concludes with the implementation details of the loss recovery algorithm itself in which the *pipe algorithm* plays the key role. In our explanation, we use the ns-3 naming convention in which all global variables are named with a prefix m_ while those without the prefix are local variables or parameter identifiers.

3.1 TCP module in ns-3

As illustrated in Figure 3, the implementation of TCP in ns-3 consists of multiple classes interacting with each other to provide reliable data transfer across the underlying network. In this section, we give a brief discussion of the main TCP classes from which our implementation is extended. Detailed explanation on them can be found in [2].

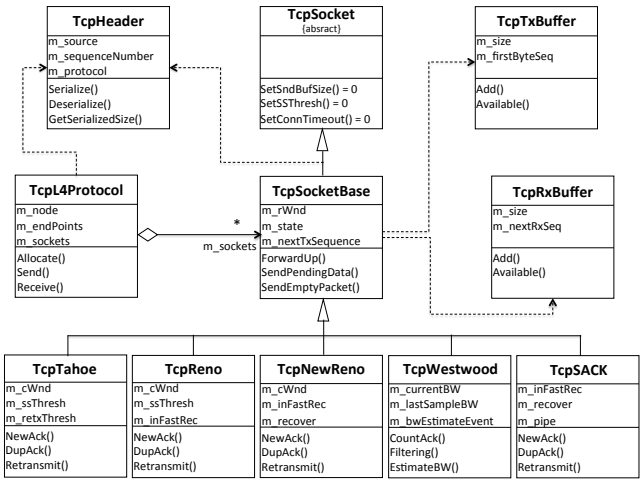

Figure 3: Class diagram of TCP module in ns-3

- `TcpSocketBase`: This class implements the main TCP features including connection management and flow control, and provides a sockets interface to be called by the upper application layer. Inherited from `TcpSocket`, this class is the base class for all TCP variant implementations.
- `TcpSocket`: This is an abstract class that contains pure virtual functions used to set essential TCP socket attributes.
- `TcpHeader`: This class implements the TCP header, including a recent extension to handle TCP options.
- `TcpTxBuffer`: This class implements a sending buffer used by TCP to hold data received from an application before transmitting.
- `TcpRxBuffer`: This class implements a receiving buffer used to store received data for reordering before passing up to the application. The class is extended in our implementation with the additional functionality of extracting contiguous, but isolated chunks of data received due to losses. The information returned from this function allows the construction of SACK blocks.
- `TcpL4Protocol`: This class implements an interface between TCP socket and the network layer.

3.2 SACK and SACK-PERMITTED options

The SACK-PERMITTED and SACK options [25] are implemented as derived classes of `TcpOption`. The processing of these options are implemented in the base class `TcpSocketBase` by the two methods `ProcessOptionSackPermitted` and `ProcessOptionSack` so that other TCP variants can use the options when desired. The construction of a SACK list, which consists of multiple SACK blocks (each block containing the left and the right edge) is assisted with an auxiliary function `TcpSocketBase::ArrangeSackBlocks` to ensure that the block containing the triggered sequence number is the first one in the SACK list. Given the 40-byte TCP option limitation, when a SACK list is added to the header, every time a call to `TcpHeader::AppendOption` indicates a failure, the last SACK block will be removed from the list and the `AppendOption` will be re-called. The goal is to transmit as many SACK blocks as possible without exceeding the maximum option length.

3.3 Global variables

The global variables that we explain here will help us clarify our discussion of the scoreboard implementation and the SACK-based loss recovery algorithm in the following sections:

m_highSack: This variable defined in `TcpScoreBoard` holds the highest sequence number that has been SACKed.

m_retxThresh: This variable defined in `TcpSack` stores the number of duplicate acknowledgments required before fast retransmit is triggered. The variable is labeled **DupThresh** in the specification.

m_highTxMark: This variable inherited from `TcpSocketBase` holds the highest sequence number that has transmitted.

m_highRxt: This variable defined in `TcpSack` keeps track of the highest sequence number that has been retransmitted.

m_rescueRxt: This variable defined in `TcpSack` holds the highest sequence number that has been optimistically retransmitted. This variable is a new variable added in RFC 6675 to allow one extra retransmission in order to sustain the ACK clock, which helps preventing the retransmission timer from being triggered [6].

m_inFastRec: This boolean variable defined in `TcpSack` determines if the TCP is currently in fast recovery phase.

m_cWnd: This variable defined in `TcpSack` represents the congestion window used to control the TCP's sending rate.

m_pipe: This variable defined in `TcpSack` holds the number of data octets that are currently in transit.

3.4 The Scoreboard

The scoreboard, which is used by a TCP SACK sender to keep track of SACK information received from the other side, is implemented as a list of scoreboard entries (`TcpScoreBoardEntry`) in which each entry contains the first sequence number of a transmitted segment m_sequenceNumber and a SACK flag m_isSacked, as depicted in Figure 4. The sequence number is used as the key for iterating and searching through the scoreboard. As mentioned in RFC 6675, the scoreboard can be implemented using a different data structure, and the list structure that we use may not be the most efficient for a production kernel implementation in terms of computation complexity. However, we leave that study for future work. Our current implementation ensures that it can perform all scoreboard functions as outlined in the specification and explained below:

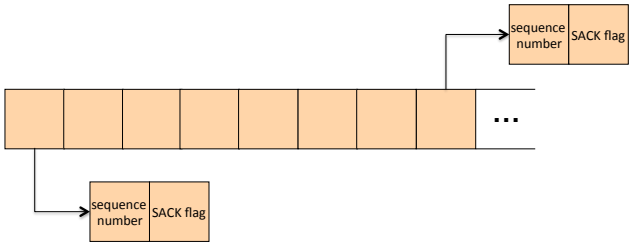

Figure 4: Scoreboard structure

`AddEntry`: This method is called by `TcpSack::SendDataPacket` when a sender sends a data segment. It creates a new scoreboard entry whose key is the initial sequence number of the transmitted segment and then pushes the entry into the scoreboard list provided that no other entry with the same key exists in the scoreboard and the scoreboard has not reached its maximum capacity.

`Update`: This method is called on the arrival of an ACK by `TcpSack::NewAck` or `TcpSack::DupAck`, and scans through the scoreboard to discard entries with sequence numbers smaller than the acknowledgment number in the received ACK segment. It then sets the SACK flags of those entries having sequence numbers reported in the SACK blocks. By simply discarding cumulatively acknowledged octets, we do not have to implement another ACK flag in the scoreboard as suggested by the specification. The method also updates **m_highSack**.

`IsLost`: This method determines if a sequence number **seq** is considered to be lost. To do so, it iterates through the scoreboard and counts the number of entries whose sequence numbers greater than **seq** and have been SACKed. If the count is at least equal to **m_retxThresh**, the method returns `true` confirming the lost of **seq**. Otherwise, it returns `false`.

`SetPipe`: This method called by `TcpSack::StartPipe` computes the number of outstanding segments. The scoreboard is traversed from the beginning to the end and only those entries with the SACK flag off and keys fall between **highAck** and **m_highTxMark** are considered. The highest sequence number that has been cumulatively acknowledged is held by **highAck**. If a sequence number key is not determined to be lost by `IsLost` or less than or equal to **m_highRxt**, then the corresponding segment is considered to be in transit.

`IsDupAck`: Due to the new definition of "duplicate acknowledgment" in RFC 6675, we implement this additional method to determine if a received segment that carries SACK information is a duplicate ACK. An acknowledgment is considered a duplicate if it carries a SACK block that identifies any previously unacknowledged and un-SACKed octets between **highAck** and **m_highTxMark**. This method called by `TcpAck::ReceivedAck` implements a nested *for* loop to iterate through the scoreboard and the received SACK list. If the method finds an entry with an unset SACK flag and a sequence number covered by any block in the SACK list that falls between **highAck** and **m_highTxMark**, it will return `true`. Otherwise, the method returns `false` indicating the received segment is not a duplicate ACK.

The following 3 methods implement the functionality of the `NextSeq` function described in RFC 6675. These methods determine the next should-be-transmitted sequence number:

`NextSeqPerRule1`: This method implements the first rule of `NextSeq`. The goal is to find a smallest unSACKed sequence number that has not been retransmitted but is determined to be lost by `IsLost`. While traversing through the scoreboard, the method stops at the first entry that has an unset SACK flag. It then checks to make sure that the sequence number **seq** of this entry is greater than **m_highRxt** but smaller than **m_highSack** and a call to `IsLost` on the sequence number returns `true`. If all conditions are `true`, the method returns **seq**. Otherwise, it will continue checking next entries until it reaches the end of the scoreboard.

`NextSeqPerRule3`: This method implements the third rule of `NextSeq`. It is almost identical to `NextDeqPerRule1` except that it does not call `IsLost` to determine whether the sequence number is considered lost.

`NextSeqPerRule4`: This method implements the forth rule of `NextSeq`. The goal is to find the highest outstanding unSACKed sequence number when **highAck** is greater than **m_rescueRxt**. The finding of the sequence to be retransmitted is implemented similarly to `SetPipe`.

The second rule of `NextSeq` in the specification is not implemented in `TcpScoreBoard` since its functionality can be accomplished by calling `TcpSocketBase::SendPendingData` to transmit a new data segment.

3.5 SACK-based loss recovery algorithm

The implementation of the SACK-based loss recovery algorithm is contained in `TcpSack`, which is inherited from the base class `TcpSocketBase` and covers the following 3 main methods:

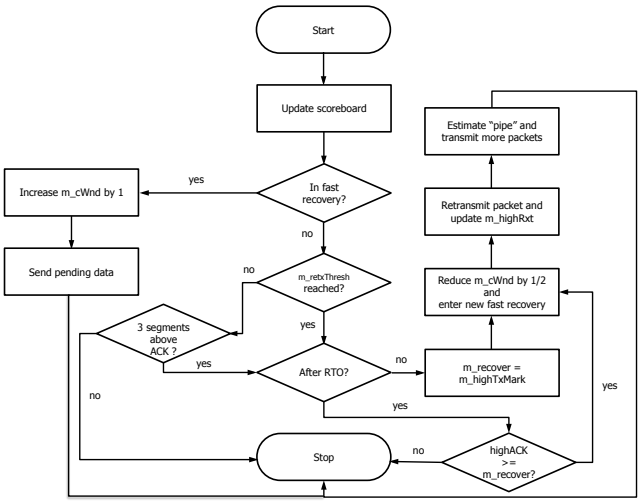

Figure 5: The SackDupAck flowchart

`SackDupAck`: This method is named to differentiate it from the `DupAck` method inherited from `TcpSocketBase` that is not implemented in `TcpSack` due to the new definition of duplicate acknowledgment in RFC 6675. The method is called by `TcpSack::ReceivedAck` if `TcpScoreBoard::IsDupAck` returns `true` indicating the received segment is a duplicate ACK. This method then calls an update on the scoreboard. If TCP is not currently in fast recovery (**m_inFastRec** is `false`) and the number of duplicate ACKs is at least equal to **m_retxThresh**, similar to Reno and NewReno, TCP enters loss recovery, halves **m_cWnd**, and retransmits the missing segment. TCP SACK then uses its pipe algorithm to take advantage of additional available congestion window for more data transmission. In case the retransmission threshold **m_retxThresh** has not reached, but a call to `TcpScoreBoard::IsLost` on the current cumulative acknowledgement point returns `true`, TCP still enters loss recovery phase and performs the same operations as in the former case. Figure 5 depicts the flowchart of the `SackDupAck` method.

`NewAck`: As illustrated in Figure 6, TCP SACK behavior on the receipt of a new ACK is very similar to NewReno except the employment of the pipe algorithm when a partial ACK is received.

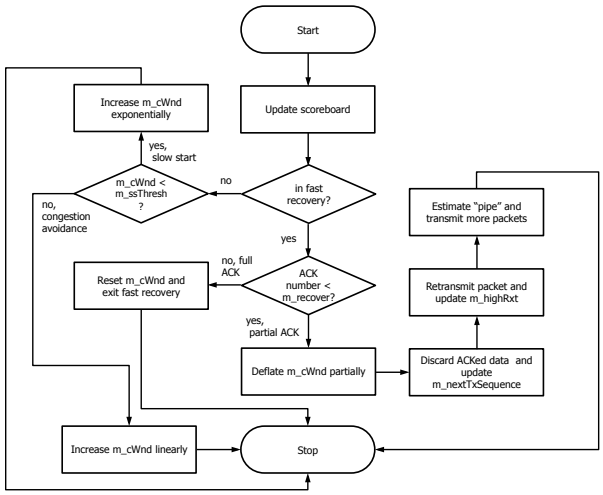

Figure 6: The NewAck flowchart

`StartPipe`: This method as depicted in Figure 7 allows TCP SACK to better utilize the available network's bandwidth using the SACK information it has been receiving from the other side. The amount of additional data that can be transmitted is the difference between **m_cWnd** and the number of outstanding octets **m_pipe** calculated by `TcpScoreBoard::SetPipe`. The octets that should be transmitted (or retransmitted) are determined using the 4 rules of `NextSeq` discussed in the previous section.

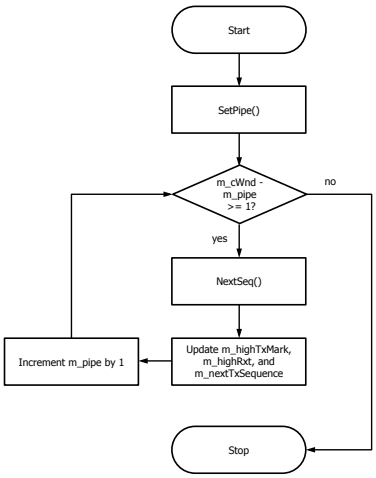

Figure 7: The StartPipe flowchart

3.6 Implementation Verification

To verify the correctness of our implementation and ensure its compatibility with existing models in ns-3, we follow the ns-3 code development guidelines by implementing a test suite `SackTestSuite` and extending the existing `TcpOptionTestSuite` with an additional test case for testing the SACK option. The TCP SACK implementation is tested using the `Ns3TcpLossTestSuite`.

4. SIMULATION ANALYSIS

In this section, we present the results obtained from simulating TCP SACK under correlated losses and our analysis of the protocol's behavior when comparing with existing TCP variants in ns-3 including Tahoe, Reno, NewReno, and Westwood+.

To study the impact of correlated losses on TCP SACK performance, we use a simple topology that consists of a single source and a single sink interconnected through two gateways as depicted in Figure 8. Each access link that connects one of the endpoints with a gateway has a capacity of 10 Mb/s and a negligible propagation delay of 0.1 ms. The bottleneck link that connects the two gateways has capacity ranging from 4 to 8 Mb/s with delay is varied from 50 to 250 ms (equivalent to 100 to 500 ms RTT). The bottleneck link is also error-prone with errors being introduced into the channel using the `BurstErrorModel`, which was implemented and contributed to the ns-3 community by the ResiliNets Research Group. The burst error rate ranges from 10^{-7} to 10^{-3}. The burst size ranges from 2 to 5 indicating the number of segments flagged as error per loss event. We use `BulkSendApplication` to generate traffic across the network with an MTU size of 1500 bytes. Each simulation has a 600 second duration and is run 10 times to achieve a 95% confidence interval. These simulation parameters are summarized in Table 1.

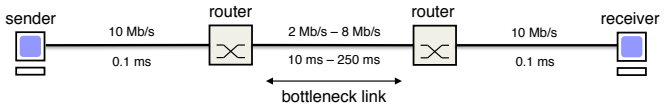

Figure 8: Single flow topology

Parameter	Values
Access link bandwidth	10 Mb/s
Bottleneck link bandwidth	4 Mb/s to 8 Mb/s
Access link propagation delay	0.1 ms
Bottleneck link propagation delay	50 ms to 250 ms
Packet MTU size	1500 B
Delayed ACK count	2 segments
Delayed ACK timeout	200 ms
Error model	BurstErrorModel
Burst error rate	10^{-7} to 10^{-3}
Burst size	2 to 5
Application type	Bulk send application
Simulation time	600 s
Number of runs	10

Table 1: Simulation parameters

Figure 9 plots the average throughput as the error rate varies from 10^{-7} to 10^{-3}. The bottleneck speed is fixed at 4 Mb/s while the bottleneck delay has a value of 50 ms. When the error rate is small, all the protocols are able to utilize the available bandwidth. However, when the error rate reaches above 10^{-5}, TCP throughput begins to drop. While SACK and NewReno perform better than Reno and Tahoe in the presence of correlated losses with high error rate, they perform worse than Westwood+ at the error rate of 10^{-3}. This is because Westwood+ does not simply reduce its congestion window when losses occur. This protocol tries to estimate

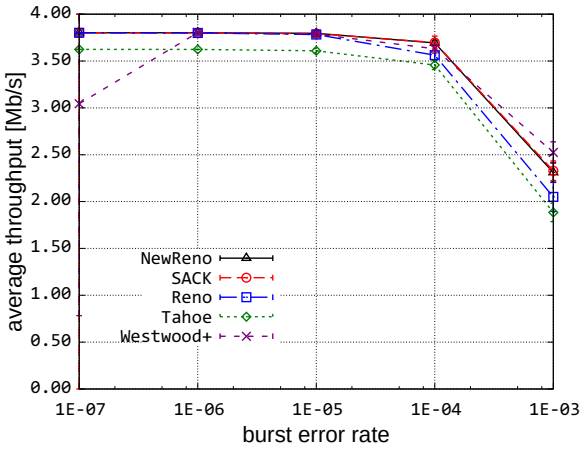

Figure 9: Average throughput vs. burst error rate

the bandwidth and uses the estimation to adjust its sending rate [14]. SACK and NewReno both perform better than Reno because they tend to take advantage of available window to recover additional losses or transmit additional new packets while they are in the loss recovery phase. Reno leaves the fast recovery phase immediately after it receives a new ACK. NewReno remains in this state until all data transmitted before it enters fast recovery is fully acknowledged. TCP SACK, using its pipe algorithm and taking advantage of information carried by SACK blocks, tries to retransmit more packet losses and transmit more data by exploiting the available congestion window whenever possible. Tahoe has the worse performance due to its returning to slow start after fast retransmit, forcing it to refill the pipe after a loss occurs.

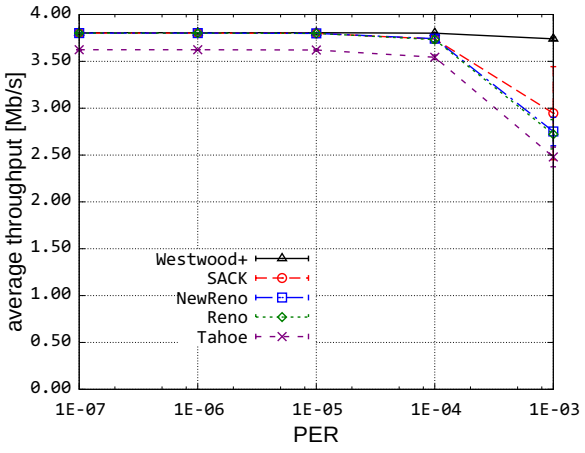

Figure 10: Average throughput vs. PER

The difference in performance between Reno and NewReno cannot be seen when losses are uncorrelated as depicted in Figure 10, which plots the average throughput against increasing PER (packet error rate). The errors in this case are generated using the `RateErrorModel` in ns-3 with packet mode. However, while NewReno performs almost identical to Reno, SACK outperforms both of them when the error rate is higher than 10^{-4}. On the other hand, the performance of Westwood+ is almost unaffected by the change in PER. Since corruption-based losses do not have an impact on network bandwidth the way congestion-based losses do, Westwood+ can achieve its throughput by using the bandwidth it estimates instead of always assuming that losses are due to congestion and reducing the sending rate like the other TCP variants.

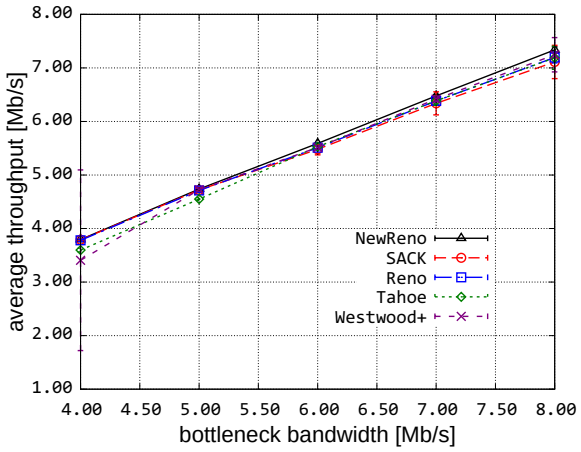

Figure 11: Average throughput vs. speed

Figure 11 plots the average throughput with respect to the bottleneck bandwidth. In this scenario, the bottleneck link has a delay of 50 ms with a capacity varying from 4 to 8 Mb/s with the burst error rate is fixed at 10^{-5}. When the bottleneck bandwidth increases, the throughput is also improved. With this small error rate, all TCP variants studied in this paper are able to utilize the available bandwidth.

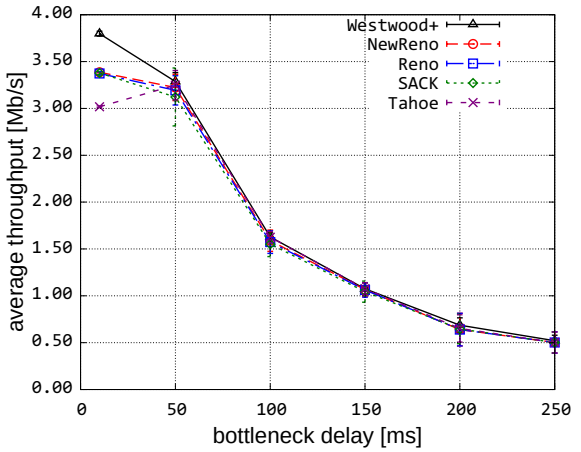

Figure 12: Average throughput vs. delay

In Figure 12, we study the impact of delay on the average throughput. While the bottleneck speed is fixed at 4 Mb/s, its delay is varied from 50 to 250 ms with the burst error rate is fixed at 10^{-5}. The performance of all TCP variants decrease significantly with the increasing in delay. Based on the previous 2 plots, we can conclude that the high delay is

the key factor of the poor TCP performance in this scenario since our error rate is too small to have any impact on the degrading throughput.

5. CONCLUSION

In this paper, we implement the TCP SACK option and TCP SACK in ns-3. Our implementations extend the ns-3 TCP module with additional models and hopefully encourages further studies and code development that are TCP-related. Our SACK option implementation allows the option to be used by any TCP variants, not only TCP SACK. We also verify our implementation with some regression tests. Finally, we study the performance of TCP SACK when it experiences multiple losses per sending window and compare it with other TCP variants that already exist in ns-3, including Tahoe, Reno, NewReno, and Westwood+. Our simulations show that when the burst error rate is higher than 10^{-5}, with a bulk transfer application, TCP SACK performs as well as NewReno and outperforms both Tahoe and Reno. With our scenarios, we do not see a noticeable difference between TCP SACK and NewReno in terms of their performance. In addition, when the error rate is low (less than or equal to 10^{-5}), all the TCP variants that we study in this paper are able to utilize the available bandwidth. Delay plays an important role in TCP performance. With a high bottleneck delay, the TCP throughput drops significantly.

For future work, we plan to extend our study by simulating TCP SACK and other variants under a higher bandwidth-×-delay network environments. We also plan to simulate these TCP protocols in a satellite environment with highly constrained bandwidth and asymmetric links. In addition, we plan to compare the different TCP acknowledgement techniques together with SACK, including NAK [13] and SNACK, a hybrid version of SACK and NAK that were developed in the context of SCPS-TP [8, 7] and employed by ResTP (Resilient Transport Protocol) [27].

6. REFERENCES

[1] The ns-3 Network Simulator. `http://www.nsnam.org`, July 2009.

[2] The ns-3 Network Simulator Doxygen Documentation. `http://www.nsnam.org/doxygen/`, July 2012.

[3] M. Allman, C. Hayes, H. Kruse, and S. Ostermann. Tcp performance over satellite links. In *Proceedings of the 5th International Conference on Telecommunication Systems*, pages 456–469, 1997.

[4] M. Allman, V. Paxson, and W. Stevens. TCP Congestion Control. RFC 2581 (Proposed Standard), Apr. 1999. Obsoleted by RFC 5681, updated by RFC 3390.

[5] E. Blanton, M. Allman, K. Fall, and L. Wang. A Conservative Selective Acknowledgment (SACK)-based Loss Recovery Algorithm for TCP. RFC 3517 (Proposed Standard), Apr. 2003.

[6] E. Blanton, M. Allman, L. Wang, I.Jarvinen, M. Kojo, and Y.Nishida. A Conservative Loss Recovery Algorithm Based on Selective Acknowledgment (SACK) for TCP. RFC 6675 (Standard), 2012.

[7] CCSDS-The Consultative Committee for Space Data Systems. Space Communications Protocol Specification (SCPS)-Transport Protocol (SCPS-TP). `http://public.ccsds.org/publications/archive/714x0b2.pdf`, October 2006.

[8] R. C. Durst, G. J. Miller, and E. J. Travis. TCP extensions for space communications. *Wireless Networks*, 3(5):389–403, October 1997.

[9] K. Fall and S. Floyd. Simulation-based comparisons of tahoe, reno and SACK TCP. *ACM SIGCOMM Computer Communication Review*, 26(3):5–21, July 1996.

[10] S. Floyd and T. Henderson. The NewReno Modification to TCP's Fast Recovery Algorithm. RFC 2582 (Experimental), Apr. 1999. Obsoleted by RFC 3782.

[11] S. Floyd, J. Mahdavi, M. Mathis, and M. Podolsky. An Extension to the Selective Acknowledgement (SACK) Option for TCP. RFC 2883 (Proposed Standard), July 2000.

[12] A. Ford, C. Raiciu, M. Handley, and O. Bonaventure. TCP Extensions for Multipath Operation with Multiple Addresses. RFC 6824 (Experimental), Jan. 2013.

[13] R. Fox. TCP big window and NAK options. RFC 1106, June 1989.

[14] S. Gangadhar, T. A. N. Nguyen, G. Umapathi, and J. P. Sterbenz. TCP Westwood Protocol Implementation in ns-3. In *Proceedings of the ICST SIMUTools Workshop on ns-3 (WNS3)*, Cannes, France, March 2013.

[15] J. L. Gil. Performance analysis of the tcp sack-based loss recovery mechanism (rfc 3517) under correlated losses. In *Proceedings of the ACM International Workshop on Performance Monitoring, Measurement, and Evaluation of Heterogeneous Wireless and Wired Networks*, PM2HW2N '06, pages 64–73, New York, NY, USA, 2006. ACM.

[16] J. C. Hoe. Start-up dynamics of TCP's congestion control and avoidance schemes. Master's thesis, University of California at Berkeley, USA, 1995.

[17] V. Jacobson. Congestion avoidance and control. *SIGCOMM Comput. Commun. Rev.*, 18(4):314–329, 1988.

[18] V. Jacobson. Modified TCP congestion avoidance algorithm, April 1990.

[19] V. Jacobson and R. Braden. TCP extensions for long-delay paths. RFC 1072, Oct. 1988. Obsoleted by RFCs 1323, 2018.

[20] V. Jacobson, R. Braden, and D. Borman. TCP Extensions for High Performance. RFC 1323 (Proposed Standard), May 1992.

[21] R. Kettimuthu and W. Allcock. Improved Selective Acknowledgment Scheme for TCP. Technical report, Argonne National Laboratory, Globus Alliance, June 2004.

[22] R. Krishnan, J. P. G. Sterbenz, W. M. Eddy, C. Partridge, and M. Allman. Explicit transport error notification (ETEN) for error-prone wireless and satellite networks. *Computer Networks*, 46(3):343–362, 2004.

[23] S. Ladha, P. Amer, J. Caro, A., and J. Iyengar. On the prevalence and evaluation of recent TCP enhancements. In *Global Telecommunications*

Conference, 2004. GLOBECOM '04. IEEE, volume 3, pages 1301–1307 Vol.3, Nov 2004.

[24] S. Mascolo, L. Grieco, R. Ferorelli, P. Camarda, and G. Piscitelli. Performance evaluation of Westwood+ TCP congestion control. *Performance Evaluation*, 55(1-2):93–111, Jan. 2004.

[25] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. TCP Selective Acknowledgment Options. RFC 2018 (Proposed Standard), Oct. 1996.

[26] P. Natarajan, N. Ekiz, E. Yilmaz, P. D. Amer, J. Iyengar, and R. Stewart. Non-Renegable Selective Acknowledgments (NR-SACKs) for SCTP. In *IEEE International Conference on Network Protocols, 2008. ICNP 2008*, pages 187–196, 2008.

[27] T. A. N. Nguyen, J. P. Rohrer, and J. P. G. Sterbenz. ResTP-A Transport Protocol for FI Resilience. In *Proceedings of the 10th International Conference on Future Internet Technologies (CFI)*, June 2015.

[28] R. Stewart. Stream Control Transmission Protocol. RFC 4960 (Standard), Sept. 2007.

[29] F. Yang and P. Amer. Non-renegable Selective Acknowledgments (NR-SACKs) for MPTCP. In *27th International Conference on Advanced Information Networking and Applications Workshops (WAINA), 2013*, pages 1113–1118, 2013.

[30] M. Zhang, B. Karp, S. Floyd, and L. Peterson. RR-TCP: a reordering-robust TCP with DSACK. In *11th IEEE International Conference on Network Protocols, 2003. Proceedings*, pages 95–106, 2003.