A SACK-based Conservative Loss Recovery Algorithm for ns-3 TCP: a Linux-inspired Proposal

Natale Patriciello
Department of Engineering Enzo Ferrari, University of Modena and Reggio Emilia
Via P. Vivarelli 10
Modena, Italy 41125
natale.patriciello@unimore.it

ABSTRACT

This paper presents a new implementation of the SACK option and RFC 6675 loss recovery algorithm in ns-3. As is already happening in the Linux kernel, we have merged the loss recovery algorithm in the part of the code shared by all TCP variants. This merge allows bringing the performance improvements under correlated losses to all congestion control algorithms. The paper then shows the performance of TCP Hybla, Highspeed, and Vegas, under a scenario with different packet error rates, comparing the performance obtained with and without the use of the TCP SACK option. The results allow concluding that the main characteristics of the congestion control algorithms are untouched, but thanks to the implemented option their resilience to correlated losses is improved.

CCS CONCEPTS

•**Networks** → **Transport protocols;**

KEYWORDS

TCP, SACK, ns-3

ACM Reference format:
Natale Patriciello. 2017. A SACK-based Conservative Loss Recovery Algorithm for ns-3 TCP: a Linux-inspired Proposal. In *Proceedings of the 2017 Workshop on ns-3, Porto, Portugal, June 2017 (WNS3 2017),* 8 pages.
DOI: http://dx.doi.org/10.1145/3067665.3067666

1 INTRODUCTION

TCP is the heart of current Internet. It still dominates regarding the number of packets and total bytes carried over the network, and many application protocols use it as the transport protocol. Nowadays, one the most important and used TCP-based protocol is HTTP (and its secure version, HTTPS) because of the increasing amount of audio and video streaming services. Therefore, it is normal that the TCP module has a lot of importance in network simulators, to correctly characterise the system's behaviour in the presence of TCP streams. With the increasing deployment of wireless mediums, satellite links, 4G and 5G mobile networks, an exciting research field is represented by TCP loss recovery. The base for the most used recovery algorithm (RFC 6675) is the SACK (Selective ACK) option extension, defined in RFC 2018. The usage of this option highly improves the TCP response to correlated losses.

In this paper, we focus our attention on ns-3, a modern open-source discrete-event network simulator because it is widely used all around the world in industry and academia. Due to the lack of SACK option and SACK-based algorithms implementations in the current mainline, we propose a new implementation of the RFC 6675 recovery algorithm on top of the TCP SACK option model given in [12]. We have taken their model for what regards the header option exchange, but we have improved their recovery loss algorithm, proposing here a new paradigm for it. For years, in literature TCP SACK has been considered a variant of TCP at the same level of TCP Cubic, Highspeed, and so on. In Linux, this difference does not exist: the SACK option is used extensively to know what segments have reached the other end, avoiding unnecessary retransmissions and precisely calculating the number of bytes in flight. The job of increasing and decreasing the congestion window remains in charge of the TCP control algorithms (Cubic, Highspeed, ...). In our implementation, we follow the design principles of the Linux stack, merging the loss recovery algorithm into the core of TCP module. In this way, ns-3 users can use SACK in conjunction with their preferred congestion control algorithm. Moreover, the implementation is designed to have one single code path for both the possibilities (SACK enabled or disabled), effectively halving the effort required to test and maintain the module. In the following, we show three different examples of TCP congestion control algorithm employed with and without the SACK option under different loss rates. We chose TCP Highspeed (RFC 3649), Hybla [2], and Vegas [1]. We decided the last two for their inherent difference: Vegas is delay-based and tries to minimize the latency, while Hybla is loss-based and proposed as a "boosted-NewReno" for improving throughput on satellite links. We would like to remark how the SACK option could be employed with success on top of these very different algorithms. We also have results concerning TCP Cubic, but we have not included them because the Cubic model is still outside the ns-3 mainline, and therefore it probably needs further work. The obtained results allow concluding that the design is improving the congestion algorithms throughput and resilience to correlated losses, without impacting their main behaviour.

The remainder of the paper is organized as follows: in Section 2 we present a survey of related work, while in Section 3 the SACK option and the recovery algorithm based on RFC 6675 are briefly discussed. Section 4 presents the implementation characteristics, and Section 5 presents the simulation setup and the analysis of

WNS3 2017, June 2017, Porto, Portugal

DOI: http://dx.doi.org/10.1145/3067665.3067666

the implementation through different examples. Finally, Section 6 concludes the paper with directions for future work.

2 RELATED WORK

The performance of TCP with and without SACK option have been widely investigated in the past. For instance, in [14] the authors presented analytic models to estimate the latency and steady-state throughput of TCP Tahoe, Reno and SACK. They validated the results using both simulations and TCP traces collected from the Internet. They showed that with independent losses, SACK performed better than Tahoe and Reno and as losses become correlated, Tahoe outperforms both Reno and SACK. Same conclusions are drawn in [16], but using a general analytical framework. It is interesting to note that often TCP SACK is considered a variant of TCP, in the same way as Vegas, New Reno, and others. In most cases, such as [9], this wrong categorization derives from the use of the simulator ns-2, in which the model of TCP SACK was separate from the other models [15]. In fact, as Linux has demonstrated, it is possible to utilise the information carried by the SACK option in all the congestion controls, as we will detail later in the paper. Moreover, problems related to the absence of SACK option have been analysed in Software Defined Networking and emergency environments where reordering and losses are two of the most critical issues [3, 4, 8].

Researchers have also investigated possible modifications to the SACK option. There is the DSACK (Duplicate-SACK) mechanism to allow the reporting of duplicate segments caused by network replication, packet reordering, ACK losses, or a premature RTO (RFC 2883). DSACK is currently employed in RR-TCP (Reordering-Robust TCP) to improve the throughput in networks suffering significant packet reordering [17]. As happened before with the SACK option, we believe that the improvements carried by RR-TCP could also be applied to other congestion control algorithms. ISACK (Improved SACK) [10] is another SACK modification to improve bit efficiency, one of the main drawbacks of SACK.

For what concerns TCP SACK and ns-3, the closest work to ours is [12], where the authors propose an implementation of TCP SACK, based on the old version of the TCP module, which is close to what was happening with ns-2: they implemented TCP SACK as another congestion control variant. As we already said, this view highly limits the correspondence between the simulator and the reality, and the performance improvements of the SACK option are not extended to the other congestion control algorithms. The class TcpSack is a derived class of TcpSocketBase and uses the implementation of a scoreboard contained in the class TcpScoreBoard. The scoreboard, which the sender uses to keep track of SACK information received, is implemented as a list of entries in which each entry contains the sequence number of a transmitted segment and a boolean flag to indicate its status. Many methods from TcpSocketBase have been reimplemented to follow the specifications of RFC 6675. In our work, we started from the new and modular TCP module, and we implemented the SACK option management to be used by all congestion control algorithms.

3 OPERATING PRINCIPLES

In 1988 V. Jacobson and R. Braden authored RFC 1072, which included the specification for a selective acknowledgments extension to TCP. They proposed it as an improvement over long-delay paths, together with Window Scale and Timestamp. Mathis, Mahdavi, Floyd, and Romanow updated and split TCP SACK specification from the others, giving its importance. The result is RFC 2018 (October 1996). It contains the definition of the SACK-permitted and SACK option format, as well as the process for generating and interpreting them in the endpoints. The document, however, did not contain any additional detail about the behavior of the TCP congestion control when coupled with SACK. These details were then published in RFC 3517 (April 2003), outdated by RFC 6675 (August 2012).

The interested reader can refer to these documents for a detailed explanation of the operating principles. In the following, we will report few points to give a quick and overall idea. The endpoints exchange the SACK-permitted option in the SYN segment to indicate that the peers can use the SACK option after the establishment of the connection. If both agree, the receiver can include a SACK option into an ACK segment addressed to the sender. The objective is to carry extended information about the acknowledgment itself. In particular, the information is about the out-of-sequence blocks of data received and then queued: in this way, the sender exactly knows what blocks it should retransmit and which are correctly received, avoiding unnecessary retransmissions. When the receiver gets the missing segments, it naturally acknowledges the data, as well as when it receives the data in continuous blocks.

From the previous short description, it should be clear that the SACK option is included only in the so-called duplicate acknowledgments. These are critical because, after the reception of three of them, the TCP performs a fast retransmit of the segment at the left edge of the window, and then enters the phase called Fast Recovery, which governs the data flow until a non-duplicate acknowledgment arrives. The RFC 6675 defines a recovery algorithm that, differently from RFC 5681, takes into account the SACKed blocks of data. It consists of a data structure, called scoreboard, and a control flow to utilize the SACK information. The main differences are (i) the TCP does not retransmit the segments marked as correctly received in the SACK block and (ii) the TCP does not consider the correctly received blocks in the count of outstanding bytes.

In practice, these objectives are reached through a combination of functions. The first is the Update () function, which keeps track of the incoming SACK options and marks accordingly the octets in the scoreboard structure. Then, we have the IsLost () function, which is responsible for declaring if a given sequence number is lost. The routine returns true when three (or the chosen value of the duplicate ACK threshold) discontiguous SACKed sequences have arrived above the given sequence. On top of the IsLost () function is built the routine that estimates the number of segments in flight. It traverses the sequence space, and for each segment, it increases the number if IsLost () return false or the segment has been retransmitted. The last function is NextSeg (), which in the recovery phase returns the next sequence number to transmit. It uses the scoreboard data structure to determine what to send following

four different rules. For space constraint, we do not report all the algorithm details.

These differences avoid unnecessary retransmissions and allow to send new data for each received block. Overall, SACK option and the previous recovery algorithm make TCP way more efficient when there are multiple losses inside a single window of transmission.

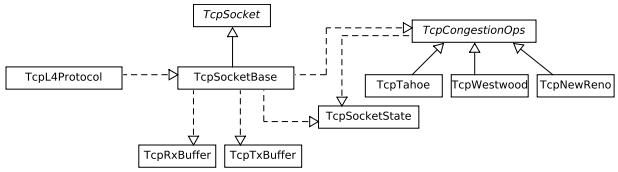

Figure 1: ns-3 TCP class diagram

4 IMPLEMENTATION

4.1 Brief Description of the Linux Implementation

The Linux implementation has support for both SACK and Duplicate SACK (DSACK), an extension of SACK. When both the original and retransmission reach the receiver, it generates a duplicate of the previous SACK option. When the other endpoint gets it, it indicates a false enter into the fast retransmission/recovery phase, because the packet got delayed in the network for excessive reordering. In the case of DSACK reception, it is possible to recover fastly from the incorrect phase transition.

Linux developers have also created the implementation of SACK-based loss recovery algorithm to be independent of the congestion control used. It means that any algorithm (Cubic, Veno, Yeah, and so on) are automatically using the SACK option to recover from losses if available. Another significant difference from RFC 6675 is that the developers have defined a smart way to integrate the two distinct code-path followed in case SACK option is enabled or disabled. By reading the RFC 6675, it is clear that its authors consider the case in which the endpoints disable the option as a fall-back to the NewReno recovery algorithm (RFC 5681). Implementation-wise, it means to have two distinct code-path, doubling the number of required line of code and, naturally, bugs. In Linux, the developers emulate the reception of SACKs for SACK-less connection in a very easy way. Without the option engaged, the receiver must send a duplicate acknowledgement each time a segment is received out-of-order. If the sender is in loss recovery phase, and it receives a duplicate ACK, it infers that the other end has correctly received a segment. Without SACK it is impossible to know what part has reached the endpoint, and therefore the sender assumes that it is the block right after the head of the outstanding window, an assumption supported by RFC 5681. For making the example clearer, let us consider a block size of 1000 bytes. If the sender receives an ACK of 5000, and then two (duplicated) ACK of 5000, it assumes that the network has lost the segment 5000-6000, while it has correctly delivered the block 6000-7000. For each received duplicated ACK, the ghostly-SACKed block moves to the right of the outstanding window (i.e., for the second one the sender assumes the network delivered block 7000-8000, and so on). It is, therefore, easy for the sender to generate SACK blocks for each duplicated ACK that follow the previous rule. In this way, emulating the reception of SACK blocks even if the option is not engaged is not improving the performance of the loss recovery algorithm (which is the same as RFC 5681), but allows to maintain only the SACK-enabled code path.

In the New Reno recovery algorithm, the partial acknowledgement mechanism allows recovering from in-window losses. These partial ACKs signal an in-window loss, and they should be correctly handled by the emulation code to avoid conflict. Practically, for each partial ACK, the SACKed bit of the outstanding window head should be cleared and then retransmitted, because the other endpoint did not correctly receive it. Moving to the previous example, after five duplicated ACKs the sender gets an acknowledgement of 8000, which is a partial ACK. It means that the receiver has got the blocks from 5000 to 8000 bytes, but the network has then lost the block 8000-9000. The sender has incorrectly assumed that the network correctly delivered it (but it was not an error because the sender has no way to know exactly which blocks are received). To fix the situation, it should move the outstanding window to the beginning of the block 8000, clearing its SACKed bit, and then retransmitting that block. At this point, if the sender receives other duplicated ACKs, they are processed as before, creating the corresponding SACK blocks.

4.2 ns-3 TCP Module Overview

In ns-3, the Internet module includes the TCP code (along with IPv4, IPv6 and UDP). It was updated during the Google Summer of Code (GSoC) 2015. In older releases, the code considered a congestion control as a stand-alone TCP. In software-engineering terms, the inheritance relation between a TCP congestion algorithm (e.g., NewReno) and the main TCP class TcpSocketBase, logically implied that "TcpNewReno was-a TcpSocketBase". The primary design principle that led the development during the GSoC was to revert such paradigm in a more sound statement: "TcpSocketBase has TcpNewReno as congestion control algorithm", which translates in avoiding the inheritance relation between these classes, and writing an interface to exchange data between sockets and congestion control modules. A famous example of such modularity is the modularization of congestion control algorithms in the Linux kernel, from which ns-3 has taken inspiration to write the API interface. In that way, porting congestion controls between Linux and ns-3 has become comfortable, and since then many algorithms have been proposed for the inclusion in the mainline [5, 7, 11].

TCP consists of multiple classes that interact, offering a reliable stream protocol to the applications. Figure 1 reports the relations between them. The solid line represents an inheritance, while the dotted line accounts for a usage of the API. For instance, TcpSocketBase uses the class TcpSocketState to store the variables of the socket accessed by other components (such as congestion window or slow start threshold). In this way, the components have a clear interface on what they can or can not modify during the life cycle of the socket.

TcpL4Protocol is a fundamental piece of the implementation: it performs multiplexing and demultiplexing of segments between sockets. For instance, when the IP layer receives a packet, it is passed to an unique (per-node) instance of TcpL4Protocol, which then passes it to the right socket through a lookup over all the available sockets. It can be considered the interface between the L4 socket and the network layer, and it performs time-consuming tasks such as checksum validation.

Classes TcpTxBuffer and TcpRxBuffer are responsible to maintain data coming from and directed to the application, respectively. Two important variables are delegated to them, namely the first unacknowledged sequence number (SND.UNA) and the next sequence number expected (RCV.NXT), extensively used during socket operations. The TcpHeader class represents the TCP header, and it manages the serialisation and deserialization of its content. The interested reader can find more details about the design and the implementation of the TCP module in [6].

4.3 ns-3 SACK Option and Recovery Algorithm Implementation

For avoiding code duplication and the effort of maintaining two different versions of the TCP core, this implementation follows the principles firstly employed in Linux. If the receiver supports SACK, the sender bases its retransmissions over the received option. However, in the absence of that option, the sender will emulate a SACK option that will trigger a reaction compliant with the RFC 5681 specification (New Reno Fast Retransmit/Recovery). The generation of such option is straightforward and follows the principles reported before and used in the Linux implementation.

The endpoints exchange the SACK-permitted option in the SYN segments only in the case that they have the "Sack" attribute in the class TcpSocketBase set to "true". In this case, the receiver maintains an ordered list of received blocks and is responsible for generating the SACK options in case an out-of-order segment is received. The class TcpRxBuffer creates the SACK option when requested by the class TcpSocketBase, following the specifications of RFC 2018. It then fills the option with as many distinct blocks as possible, without exceeding the limit of the TCP header size.

The class TcpTxBuffer is responsible for maintaining an updated scoreboard for the sender. It is updated each time the sender receives a SACK option (from the other endpoint or emulated) through the method Update (), defined in RFC 6675, that updates the flags for each transmitted block in the transmission queue. In TcpTxBuffer we have also implemented others methods defined in the RFC, such as NextSeg (), which indicates the next block to send by keeping into considerations the SACKed blocks, and IsLost (). We also have a method for estimating the current number of outstanding bytes, that uses IsLost () to determine if a particular block has been lost.

It is worth to note that, in the case of a Retransmission Timeout (RTO) expiration, the code reset all the information about the SACKed blocks. Therefore, the implementation will re-send all segments starting from SND.UNA, even the ones correctly received.

During the development, we have put a strong focus on the testing strategies. In the history of ns-3, TCP-based testing was done through the comparison of the results of a simulation to a reference trace files. Obviously, this method is not sustainable at all, because it involves many components, and any change in them can determine a different output. Instrumental to understanding this, after the introduction of the Timestamp option the ns-3 team had to re-generate all the trace files which involved TCP, because of the presence of additional values in the options part of the TCP header. Instead, we have used a design in which it is possible to test every single added function, especially in TcpTxBuffer that represents the scoreboard data structure. The testing is done through explicit calls to a TcpTxBuffer instance, to ensure a correct response from the class itself by comparing the output of these functions with expected values. Other aspects of the TCP socket functionalities are already tested through a particular testing framework, introduced after the 2015 GSoC and still in use.

Figure 2: Examples environment

5 NS-3 SACK AT WORK

In this section, we present different runs of an example program to show the SACK behaviour. Our objective is not to show the performance improvements of the SACK option in case of correlated or uncorrelated losses, as there are plenty examples available in the literature; instead, we will point out what is effectively happening on a connection when the option is or is not employed. As a side effect, the graphs will demonstrate how SACK information influences the TCP behaviour, proving the validity of the implementation. The code used for the following experiments is available in the examples/tcp directory of [13], specifically the tcp-variant-comparison example.

We make use of a simple topology, consisting of a sink/source pair interconnected through a gateway, as we can see in Figure 2. There is no need to overcomplicate the topology, in this case, giving our objective. In the following, we report the fixed parameter for each simulation, listed also in Table 1. The access link that connects the source endpoint with the gateway has a capacity of 100 Mb/s and a propagation delay of 5 ms. The bottleneck link that connects the gateway and the sink has a capacity of 10 Mb/s and a delay of 45 ms. The bottleneck link also has an error model. To generate correlated losses, we have used BurstErrorModel, with a burst size that is variable between two and five, which represents the number of packets dropped in a loss event. We used BulkSendApplication to generate the traffic, with an MTU size of 1500 bytes. Each simulation has a 100-second duration. Other parameters, such as the TCP congestion control algorithm, the presence of the SACK option, and the burst error rate, are different for each experiment and we will indicate these values in the description of each test. We vary the burst error rate from 10^{-2} to 10^{-4}. The values are chosen indicatively to show a range that starts from a very high, to an average, error rate. We randomly pair different congestion control algorithms to different error rates to show their effects on congestion window evolution and sequence number transmitted.

Table 1: Fixed simulation parameters

Parameter	Value
Access Bandwidth	100 Mb/s
Access Delay	5 ms
Bottleneck Bandwidth	10 Mb/s
Bottleneck Delay	45 ms
MTU	1500 B
Delayed ACK count	2 segments
Delayed ACK timeout	200 ms
Error model	Burst
Burst size	2 to 5 packets
Application type	BulkSend
Simulation time	100 s

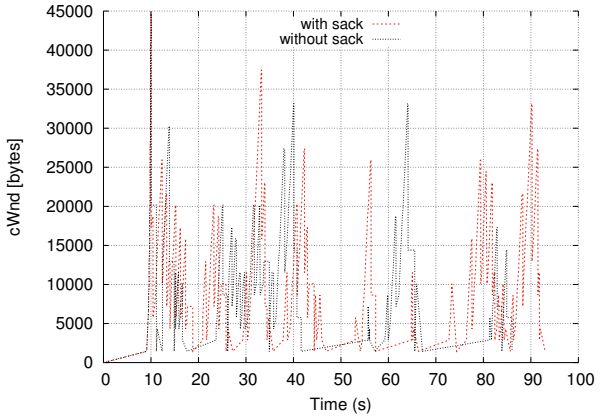

Figure 3: TCP Highspeed congestion window, very high error rate (10^{-2})

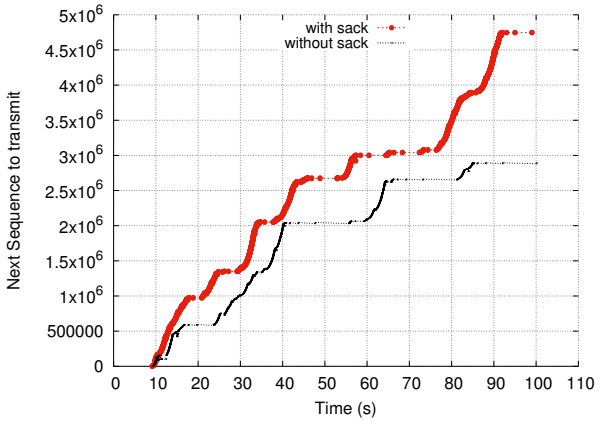

Figure 4: TCP Highspeed transmitted sequence numbers, very high error rate (10^{-2})

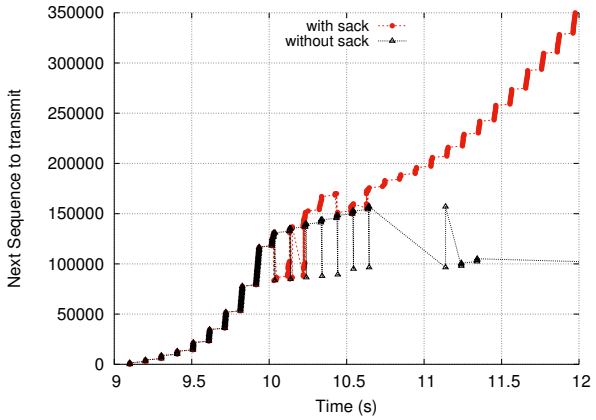

Figure 5: TCP Highspeed sent sequence numbers, particular with high error rate (10^{-2})

5.1 TCP Highspeed and SACK

As the first example, let us take the Figure 3 in which we depicted the congestion window evolution for TCP Highspeed, in two different runs, with and without SACK. The very high burst error rate makes the congestion window growth very similar. In the presence of the option, the congestion control still increase and decrease the window as per RFC 3649. Near the end of the simulation, at the 85th second, we can see that the black line (which represents the transmission without the SACK option) suddenly stops. It means that the high number of drops has interrupted the communication since after six retransmission attempts the TCP stops trying and abort the connection itself. With SACK this problem is mitigated because it does fewer retransmissions (thanks to its principles) and there is a lower probability that a retransmitted segment is lost. For the same experiment, we plot in Figure 4 the transmitted sequence numbers. It is easy to see from the different shapes of the curves that when the implementation uses the SACK option, it can transmit a greater number of segments. Raw values of the sequence numbers demonstrate such fact: with SACK, the application sends 5 MB, while without SACK it can only transmit a bit less than 3 MB.

To better analyse the behaviour with the SACK option, we have reported in Figure 5 what is happening between the 9 and the 12 second of simulated time. The red points show the sequence number transmitted by the application when using SACK, while the triangles show the sequence numbers sent without using SACK (the line connecting them is put to have also a logic connection between them). Until the 10th second of simulation, the two behaves the same. Then, at 10.1 s, there is a burst of losses: without the SACK, the implementation retransmits only the first of these segments (one single triangle). On the other hand, we can see that there are many red points in that area: it means that the TCP is retransmitting the burst of packets lost, thanks to the SACK information that comes from the other endpoint. The application without SACK then uses the partial acknowledgement mechanism to try to recover from losses, while sending new data. The TCP sends some data out (the triangles on top), then it receives a partial ACK

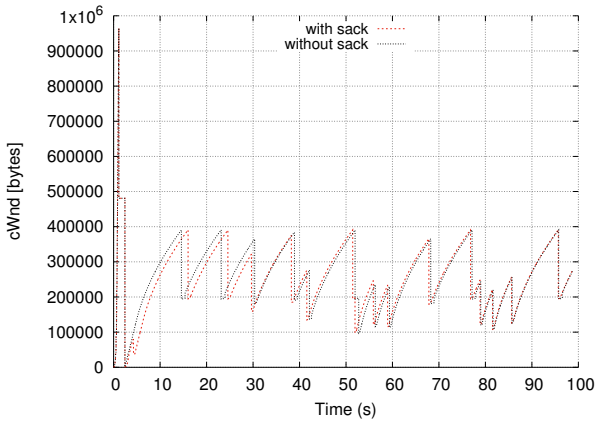

Figure 6: TCP Hybla congestion window, average error rate (10^{-4})

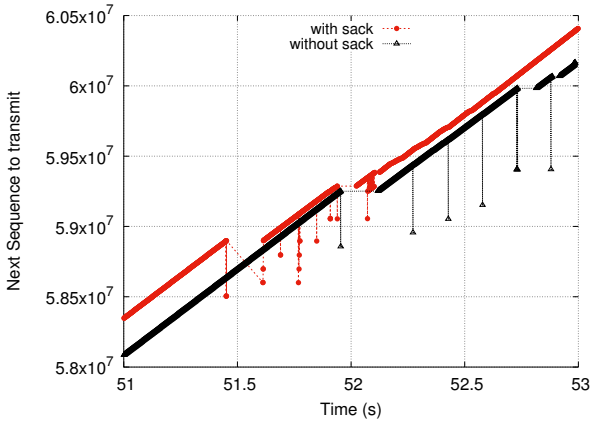

Figure 7: TCP Hybla transmitted sequence numbers, particular at average error rate (10^{-4})

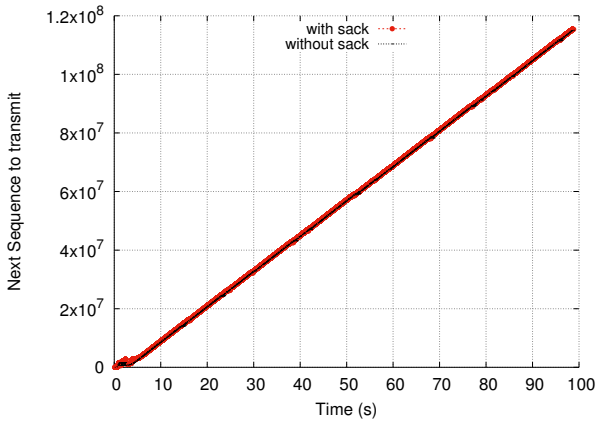

Figure 8: TCP Hybla transmitted sequence numbers, average error rate (10^{-4})

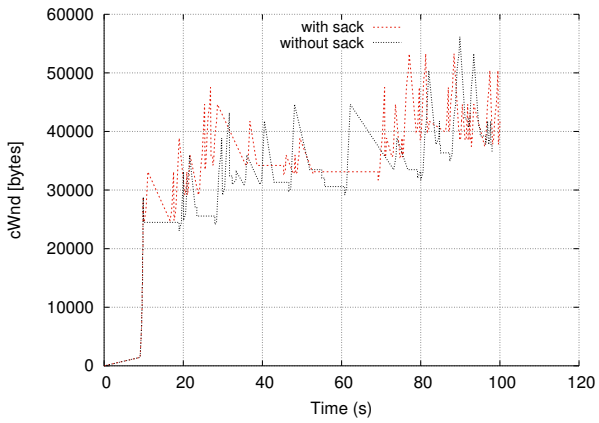

Figure 9: TCP Vegas congestion window, high error rate (10^{-3})

and retransmit a single segment (a single, lower triangle) until it gets a timeout, and reset the next sequence near the values at the bottom (11.3 seconds of simulated time). The same does not apply for TCP that uses the SACK information: it can recover quickly and can send more data avoiding the retransmission of already received blocks. Thanks to this fact, it does not encounter a timeout.

5.2 TCP Hybla and SACK

In Figure 6 we have reported the congestion window evolution for TCP Hybla, patched to introduce a pacing system that limits to two the segments sent for each incoming ACK. The first thing that we can notice is that the slope of the two curves is very similar, except for a 20-second window at the beginning of the simulation. The very same shape also indicates that the presence (or the absence) of the option, does not influence the congestion control algorithm fundamentals. TCP Hybla is still working as expected when increasing the window. In the end, the losses are well recovered even without the SACK option in place, probably through the partial ACK mechanism inherited from the New Reno algorithm.

Figure 7, in which we report a particular of the plot of sequence number transmitted, supports the previous statement. With SACK there is a short period in which the TCP retransmits the lost segment (from 51.4 s to 52.2 s) while without it the retransmissions are done in a more wide time span. The retransmissions indicate that, on one hand, the continuous reception of the SACK blocks, while in the other the arrival of spaced partial ACKs. The complete picture of the transmitted sequence is reported in Figure 8, and it confirms that not using the SACK option leads to the same performance of the SACK-enabled version. However, this conclusion is valid only in this scenario with this specific error rate: it is evident that, increasing the error rate or introducing packet reordering in the script, worses the performance of Hybla without SACK.

5.3 TCP Vegas and SACK

Another interesting comparison is between a TCP Vegas and TCP Vegas + SACK. We reported the congestion window evolution in

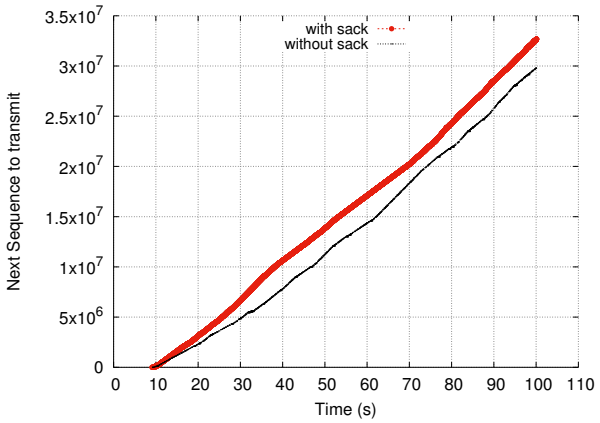

Figure 10: TCP Vegas transmitted sequence numbers, high error rate (10^{-3})

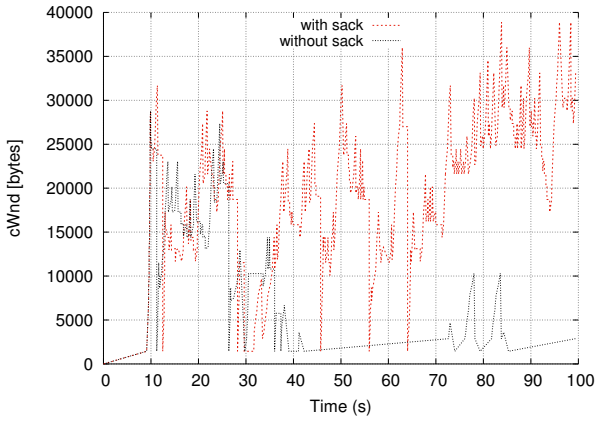

Figure 11: TCP Vegas congestion window, very high error rate (10^{-2})

Figure 9, in an environment with a 10^{-3} burst error rate. It is evident the better performance of the SACK-enabled version, supported by higher values of the congestion window. Since the Vegas algorithm is delay-based, the values of the congestion window depends on the round trip time measured at the sender. Higher values imply that the estimated time is lower when employing the SACK option. Giving that the delay of the channel is fixed, this is a clear signal of the fact that there are fewer packets in the queues, reducing the overall queuing delay of the network. Thanks to the higher value of congestion window, the TCP can transmit more data (Figure 10). When increasing the error rate to 10^{-2}, the benefits of the SACK option on TCP Vegas becomes very pronounced, as we can see in Figure 11. Without the option, the congestion window values are very limited, inevitably affecting the throughput.

6 CONCLUSIONS

In this paper, we described a new implementation of the TCP SACK option, and the loss recovery algorithm outlined in the RFC 6675, in ns-3. The most important characteristic of the loss recovery algorithm is the possibility to apply the performance improvements of RFC 6675 to any congestion control, as it is already happening in the Linux kernel. Another important feature is that, thanks to the SACK emulation process, we are able to maintain only a single code path even if the SACK option is disabled. We have therefore demonstrated through simulations that TCP variants such as Hybla, Highspeed, and Vegas, could use the SACK information to improve the performance in environments with correlated losses while maintaining their characteristics. As future work, we plan to extend the SACK implementation by adding the capability to process Duplicated SACK and to improve its memory utilization, as well as lowering its computational complexity. Moreover, it is planned to do a comparison with a trace obtained through the use of DCE and a real Linux kernel.

ACKNOWLEDGMENTS

The author would like to thank Carlo A. Grazia, Martin Klapez, and Maurizio Casoni for their cooperation and valuable contribution.

REFERENCES

[1] L. Brakmo and L. Peterson. 1995. TCP Vegas: End to End Congestion Avoidance on a Global Internet. *IEEE Journal on selected Areas in communications* 13, 8 (1995), 1465–1480.

[2] C. Caini and R. Firrincieli. 2004. TCP Hybla: a TCP Enhancement for Heterogeneous Networks. *International journal of satellite communications and networking* 22, 5 (2004), 547–566.

[3] M. Casoni, C.A. Grazia, and M. Klapez. 2016. Enabling Resource Pooling in Wireless Networks Through Software-defined Orchestration. *2016 IEEE International Conference on Communications Workshops, ICC 2016* (2016), 730–735. DOI: http://dx.doi.org/10.1109/ICCW.2016.7503874

[4] M. Casoni, C.A. Grazia, and M. Klapez. 2016. An SDN and CPS Based Opportunistic Upload Splitting for Mobile Users. *Lecture Notes of the Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering, LNICST* 169 (2016), 67–76. DOI: http://dx.doi.org/10.1007/978-3-319-47063-4_6

[5] M. Casoni, C. A. Grazia, M. Klapez, and N. Patriciello. 2015. Implementation and Validation of TCP Options and Congestion Control Algorithms for ns-3. In *Proceedings of the 2015 Workshop on ns-3 (WNS3 '15)*. ACM, New York, NY, USA, 112–119. DOI: http://dx.doi.org/10.1145/2756509.2756518

[6] M. Casoni and N. Patriciello. 2016. Next-Generation TCP for ns-3 Simulator. *Simulation Modelling Practice and Theory* 66 (2016), 81–93.

[7] C.A. Grazia, M. Klapez, N. Patriciello, and M. Casoni. 2015. PINK: Proactive INjection into acK, a Queue Manager to Impose Fair Resource Allocation among TCP Flows. *2015 IEEE 11th International Conference on Wireless and Mobile Computing, Networking and Communications, WiMob 2015* (2015), 132–137. DOI: http://dx.doi.org/10.1109/WiMOB.2015.7347952

[8] C.A. Grazia, M. Klapez, N. Patriciello, M. Casoni, H. Gierszal, P. Tyczka, K. Pawlina, A. Amditis, and E. Sdongos. 2015. Performance Evaluation and Economic Modelling of PPDR Communication Systems. *2015 IEEE 11th International Conference on Wireless and Mobile Computing, Networking and Communications, WiMob 2015* (2015), 75–82. DOI: http://dx.doi.org/10.1109/WiMOB.2015.7347943

[9] M. Kazmi, A. Shamim, N. Wahab, and F. Anwar. 2014. Comparison of TCP Tahoe, Reno, New Reno, Sack and Vegas in IP and MPLS Networks under Constant Bit Rate Traffic. In *International Conference on Advanced Computational Technology and Creative Media (ICACTCM)*. 33–39.

[10] R. Kettimuthu, W. Allcock, and others. 2004. Improved Selective Acknowledgment Scheme for TCP. In *International Conference on Internet Computing*. 913–919.

[11] T. Nguyen, S. Gangadhar, M. Rahman, and J. Sterbenz. 2016. An Implementation of Scalable, Vegas, Veno, and YeAH Congestion Control Algorithms in ns-3. In *Proceedings of the Workshop on ns-3 (WNS3 '16)*. ACM, New York, NY, USA, 17–24. DOI: http://dx.doi.org/10.1145/2915371.2915386

[12] T. Nguyen and J. Sterbenz. 2015. *An Implementation of the SACK-Based Conservative Loss Recovery Algorithm for TCP in ns-3.* ITTC Technical Report ITTC-FY2015-TR-69221-02. The University of Kansas, Lawrence, KS.

[13] N. Patriciello. 2017. ns-3 source code and scripts used for this paper. https://github.com/kronat/ns-3-dev-git/tree/sack-wns3-2017. (February 2017).

[14] B. Sikdar, S. Kalyanaraman, and K. Vastola. 2001. Analytic Models and Comparative Study of the Latency and Steady-State Throughput of TCP Tahoe, Reno

and SACK. In *IEEE 2001 Global Telecommunications Conference (GLOBECOM'01)*, Vol. 3. IEEE, 1781–1787.

[15] D. X. Wei and P. Cao. 2006. ns-2 TCP-Linux: An ns-2 TCP Implementation with Congestion Control Algorithms from Linux. In *The 2006 Workshop on ns-2: The IP Network Simulator*. ACM, New York, NY, USA, Article 9. DOI: http://dx.doi.org/10.1145/1190455.1190463

[16] A. Wierman and T. Osogami. 2003. A Unified Framework for Modeling TCP-Vegas, TCP-SACK, and TCP-Reno. In *Proceedings of the 11th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer Telecommunications Systems (MASCOTS 2003)*. IEEE, 269–278.

[17] M. Zhang, B. Karp, S. Floyd, and L. Peterson. 2003. RR-TCP: a Reordering-Robust TCP with DSACK. In *Proceedings of the 11th IEEE International Conference on Network Protocols*. IEEE, 95–106.