# Questioning the NS-3 TCP Implementations and Suggesting Areas for Improvement

Allison Hume
School of Engineering
University of California, Santa Cruz
Santa Cruz, California 95064
Email: arhume128@gmail.com

Katia Obraczka
School of Engineering
University of California, Santa Cruz
Santa Cruz, California 95064
Email: katia@soe.ucsc.edu

***Abstract*—The existing TCP implementation in the NS-3 network simulator is not sufficiently accurate to be useful for making comparisons between different congestion control algorithms. This work demonstrates how the current implementations in both the production and development versions of the simulator differ from results presented in previous literature. It is observed that errors are introduced by the lack of a consistent, extensible design for congestion control. The main contribution is a design that will make the NS-3 TCP implementation more accurate and make it easier for new congestion control algorithms to be implemented in the simulator.**

## I. Motivation

Simulators are a very important tool for the networking community. In contrast to live hardware, simulators allow experiments to cover a large number of topologies and allow researchers to experiment easily with changes in experiment or protocol settings. One critical feature that a network simulator must offer is the ability to run code that is as close to live as possible. Code should reflect the state of the art in hardware and be easily modified and expanded so that researchers who want to experiment with new protocols can easily add them to the existing set of code.

One active area of research in the TCP community is introducing and comparing different congestion control algorithms. Network simulators are essential tools for validating, understanding, and comparing these algorithms and NS-3 and its previous incarnation, NS-2, have a long track record of usage in the TCP community for this purpose [1], [7]. As the simulator has changed and evolved, however, there has not been enough of an effort to continue to validate and understand the simulator itself. The work presented in this paper began as an effort to understand a newly proposed TCP congestion control algorithm, TCP Inigo [14], using NS-3 as a simulator platform. It became clear, however, that the existing TCP implementation in NS-3, while potentially sufficient for comparing aggregate metrics such as throughput, or for supporting higher level protocols, may not be robust enough to support the detailed analysis that is necessary to understand and compare congestion control algorithms.

Figure 1 shows a trace of the congestion window for TCP Westwood using an experiment first mentioned in Gangadhar et al [6]. The plot on the left shows the experiment run on NS-3 version 3.24 while the plot on the right shows the same experiment run on NS-3 version 3.25. This figure shows that the implementation of TCP Westwood has changed drastically between the two implementations. How can researchers trust simulators to compare different versions of TCP when established protocols are not consistent between simulator versions? This work has been motivated by these findings, and by the difficulties encountered in adding a new congestion control algorithm to NS-3. The main contribution in this work is a design for simulated TCP that will be easier to maintain, and more extensible, than existing designs. The design will be presented after a brief background on TCP congestion control and existing TCP code designs. This work also presents results that question the existing TCP implementation in NS-3, lending weight to the argument that a comprehensive design evaluation, as well as formal validation process, is very much needed.

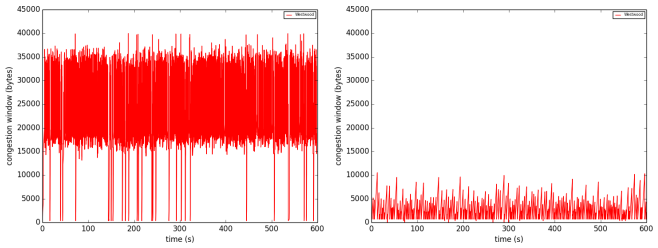


Fig. 1. A 600 second trace of the congestion window of TCP Westwood. Experiment from Gangadhar et al [6] run on both NS-3 version 3.24 and 3.25.

## II. Background

Congestion control in TCP involves the interaction of two main variables: congestion window (cwnd) and slow start threshold (ssth). The TCP sender uses cwnd to adjust its rate of transmission and ssth is used to choose the algorithm by which cwnd is computed. Originally, congestion control was defined by four algorithms: slow start, congestion avoidance, fast retransmit, and fast recovery [2]. Slow start and congestion avoidance are the two algorithms used by the sender to compute cwnd: slow start is used if cwnd is below ssthresh, and congestion avoidance otherwise. The slow start algorithm is used at the beginning of transmission, or after a congestion

event causes cwnd to fall below ssth, and increments cwnd exponentially until a congestion event occurs or ssthresh is reached. The congestion avoidance algorithm increments cwnd more slowly, until a congestion event occurs. The fast retransmit algorithm is used to detect and repair loss based on incoming duplicate ACKs and the fast recovery algorithm is used to adjust cwnd until a non-duplicate ACK arrives.

Congestion control algorithms refer to specific implementations of those four algorithms, and potentially additional components such as Selective Acknowledgements (SACK), which make it possible for the receiver to acknowledge scattered blocks of data, not just a continuous block. Not all algorithms, however, may differ between different congestion control algorithms. For example, the TCP Westwood [9] congestion control algorithm is a modification of the TCP Reno congestion control algorithm that changes only how cwnd and ssth are changed after three duplicate ACKs or a timeout. While Reno halves the congestion window after three duplicate ACKs, Westwood uses a bandwidth estimator to try to match the variables to the currently available bandwidth. The two congestion control algorithms, however, have the same implementation of slow start.

The previous discussion helps to illuminate some of the challenges that arrise when attempting to implement TCP congestion control in a way that is both efficient and extensible. Extensibility is much more important in a simulator than in hardware. While a new version of TCP congestion control will only be added to the Linux kernel, for example, rarely, new versions will be added to simulators frequently for validation and testing. If the TCP implementation in Linux, therefore, promotes efficiency over extensibility, that is ok. In a simulator, however, lack of extensibility goes directly against the goals of the software. The next section will discuss the recent history of design choices in NS-3 , before a modified design for any simulator is presented.

## III. Design

The previous NS3 release, NS-3 version 3.24 has a module called tcp-socket-base that contains the main function from which congestion control functionality is used: ReceivedAck. In that release the functionality in ReceiveAck is minimal, it servers mostly to call the NewAck and DupAck functions, which are defined for each TCP version, in its respective module. In other words, there is one main module for each TCP version containing all code that is required for that specific congestion control algorithm [10]. A visual of this design can be seen in the top left of Figure 2. The downside of this design approach is code duplication and the difficulty of adding a new congestion control algorithm, although this approach does make the implementation of specific algorithms very clear.

A recent effort has been underway to refactor the TCP implementation of NS-3 so that the majority of TCP code is shared and only specific congestion control functions must be implemented to define a new version of congestion control [12]. The result of this effort is the recently released NS-3 version 3.25 [11]. In this version, the ReceivedAck function in tcp-socket-base contains much more functionality and calls more specific, well defined congestion control functions PktsAcked, GetSsThresh, and IncreaseWindow, which are defined in a congestion-ops module as the base class for an congestion control implementation. The top right portion of Figure 2 summarizes this design. This is a much more extensible approach than the previous design, although great care must be taken to be sure that all shared code is truly generic. As will be discussed further in the results section, that is not currently the case. Another criticism of this design is that currently the functions for TCP NewReno are also implemented in congestion-control-ops, while the functions for TCP Westwood are placed in a separate module. It is not totally clear why this choice was made, although one guess is that it is because NewReno is the default TCP congestion control algorithm. This design also does not make the inheritance clear. It is not obvious, for example, that the Westwood implementation actually uses the SlowStart function defined for NewReno, since SlowStart is not defined in the base class. We believe that this design is moving in the correct direction, but that additional improvements can be made.

We propose a new design for the NS-3 TCP module that builds off the current modular approach. Our design is based off the following goals for the design of TCP Congestion Control in any simulator:

**Extensibility:** The primary goal of a simulator is to allow researchers to test and validate new ideas. For the design to support the TCP community in this way it must be straightforward to add new congestion control implementations to an existing code base.

**Modularity:** Congestion control is characterized by fact that it is made up of a set of algorithms. As such, a modular design is necessary to allow researchers to experiment with the different pieces that make up a congestion control algorithm, without having to create a completely new implementation for each set of existing components.

Based on these goals, we believe that an implementation of TCP congestion control on a simulator should be modularly built up out of components in such a way that any component is easily switched out or turned off. This would also allow different versions of congestion control to share implementations for specific algorithms. The functionality of ReceivedAck should be reduced slightly, so it does not include specific Fast Retransmit and Fast Recovery algorithms. The tcp-congestion-ops module will remain the base class for all congestion control implementations, without also containing the implementation of NewReno. In addition, the base class will be expanded to also include SlowStart, FastRetransmit, and FastRecovery algorithms. Each function in the base class will have a corresponding module in which all various implementations for those functions will be placed. Lastly, each specific congestion control algorithm will have a module that defines the selection of sub-algorithms of which it is comprised. The bottom portion of Figure 2 summarizes the

proposed design.

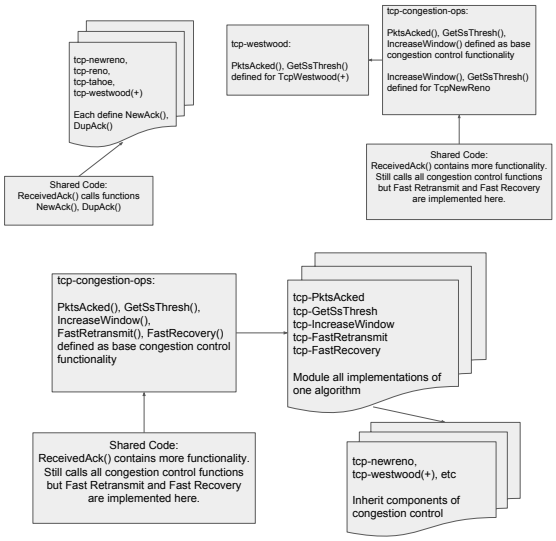


Fig. 2. Diagrams showing the TCP Congestion Control designs for NS3 versions 3.24 and 3.25 (from top left) as well as the design proposed here (bottom).

## IV. Related Work

In 2006 there was a significant effort performed by Wei and Cao [15] to improve the extensibility, validity, and performance of NS-2 by bringing the TCP implementation closer to the Linux TCP implementation. Not all of their improvements exist currently in NS-3 and this work is very similar in motivation to their work. The main difference is that this work also aims to provide a framework for future congestion control algorithms by promoting a design that allows explicit inheritance of congestion control algorithms, when new algorithms are modifications of previous algorithms.

In 2015 Casoni et al [4] performed a validation of NS-3 congestion control algorithms by comparing the operation of those algorithms to the Linux implementations. Since 2015 the NS-3 TCP implementation has been refactored, so their work is an important result for ensuring repeatability. This work attempts to go farther that verification by suggestion design changes to improve the extensibility and accuracy of the NS-3 TCP implementation.

Bateman and Bhatti [3] studied the versions of TCP congestion control that existed in the NS-2 simulator in 2010 and found that, using Jain's fairness index as a metric, there were significant variations between the simulator and their physical testbed. This work is an important precedent for questioning the validity of TCP congestion control algorithms in simulators. Instead of using the fairness index as a metric, however, this work investigates congestion window, threshold value, and queue length as comparison metrics in order to make more specific comparisons and as a starting place for identifying what aspects of the implementation cause issues to occur.

## V. Results

Different experiments show different levels of variation between the current NS3, ns3-3.25 version and previous results obtained from either NS2, previous versions of NS3, or hardware. Figure 3 shows an experiment originally presented in Casoni et al [4], who presented results comparing TCP NewReno on NS3 version ns3-3.22 to Linux. Their results showed cwnd and ssth from NS3 both very close in scale and pattern to the values obtained from Linux, although cwnd showed major spikes that were attributed to fast retransmit. Figure 34(a) shows the same experiment repeated on NS3 version ns3-3.24, with results very similar to those in the original publication. The source of the spikes, however, discovered to be the increase of cwnd by one segment for every duplicate ACK after the three duplicate ACKs that initiate fast recovery (section 3.2 of RFC 2582 [5]). Figure 34(b) shows the same experiment with that increase removed. This trace now more closely matches the Linux results presented in Casoni et al [4].

One thing that is import to note is that this difference was not the result of a bug in NS3. The code very closely follows the RFC, however the Linux TCP implementation handles fast recovery differently. In the Linux implementation cwnd holds the valid number of segments allowed to be outstanding in the network throughout fast recovery, it does not require arbitrary adjusting of the congestion window [4] [13]. The important point here is that, in order to disable fast recovery, the code for fast recovery had to be tracked down and commented out within the shared code base. The current design of TCP in NS3 allows no option for changing or disabling Fast Retransmit and Fast Recovery algorithms. In different scenarios you may want different implementations one algorithm, or, as in this case, simply to turn it off. Commenting out sections of code is a very error prone method of achieving that result.

Figure 34(c) and 4(d) show the same set of experiments, but run on NS3 ns3-3.25 version, with the refactored TCP. It is important to reiterate that, although the intention of this refactoring was to make the TCP congestion control design modular and extensible, it was still necessary to manually find and comment out the relevant lines of code corresponding to fast recovery. Although there is not a drastic difference between the results for the two versions of NS3, the new version does respond differently to the removal of the fast recovery algorithm.

The parameters of the previous experiment included a packet loss rate of $10^{-3}$. Figure 4 compares another experiment between NS3 versions 3.24 and 3.25, this time for experiments with a packet loss rate of 0. The two rows show two experiments with the left column providing results for version 3.24 and the right column providing results for version 3.25. The top row attempts to duplicate an experiment fist presented in Grieco et al [7] and seen also in Abdeljaouad et al [1]. This experiment originally included one forward flow with 10 back flows turning on and off but, due to the results discovered in attempting duplication, the experiment is

TABLE I
SUMMARY OF EXPERIMENT PARAMETERS

| Origin | Figure | Bottleneck Link | Access Link | MTU Size | Rate of Packet Loss |
|---|---|---|---|---|---|
| Gangadhar et al [6] | Figure 1 | 2 Mbps, 0.01ms delay | 10 Mbps, 45ms delay | 400 B | $5 * 10^{-3}$ |
| (tcp-variants-comparison in NS-3 examples) | Figure 4 | | | | 0 |
| Casoni et al [4] | Figure 3 | 10 Mbps, 25ms delay | 100Mbps, 1ns delay | 1500 B | $10^{-3}$ |
| Grieco et al [7] | Figure 4 | 2Mbps, 125ms delay | 1Gbps, 0.01ms delay | 1500 B | 0 |

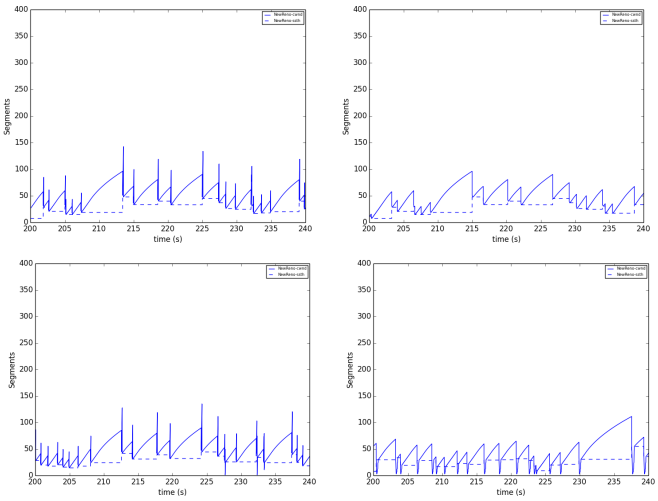


Fig. 3. 40 seconds of a 600 second trace show congestion window and slow start threshold for NewReno in ns3-3.24 (top row) and ns3-3.25 (bottom row), with fast recovery on (left column) and off (right column).

presented here without the backflows.

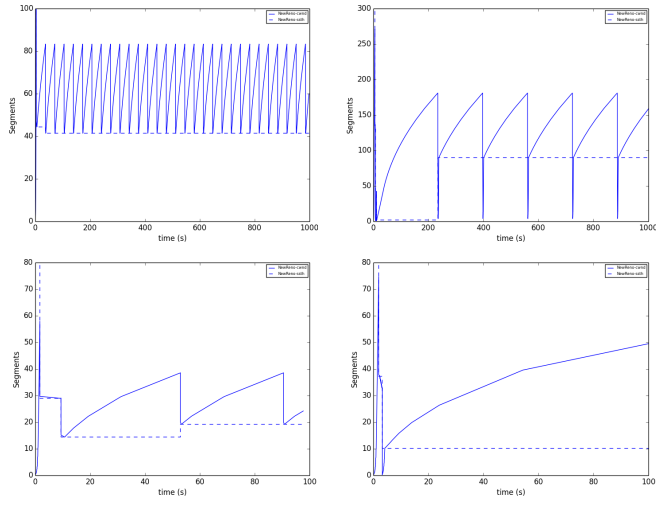


Fig. 4.

## VI. Discussion and Future Work

## VII. Conclusion

## VIII. Acknowledgements

Thank you to Andrew Shewmaker for the initial research direction, information about TCP refactoring in NS-3, and help with interpreting results.

## References

[1] I Abdeljaouad, H Rachidi, S Fernandes, and A Karmouch. Performance analysis of modern tcp variants: A comparison of cubic, compound and new reno. In *Communications (QBSC), 2010 25th Biennial Symposium on*, pages 80–83. IEEE, 2010.

[2] M Allman, V Paxson, and E Blanton. Tcp congestion control. *RFC5681*, 2009.

[3] Martin Bateman and Saleem Bhatti. Tcp testing: How well does ns2 match reality? In *Advanced Information Networking and Applications (AINA), 2010 24th IEEE International Conference on*, pages 276–284. IEEE, 2010.

[4] Maurizio Casoni, Carlo Augusto Grazia, Martin Klapez, and Natale Patriciello. Implementation and validation of tcp options and congestion control algorithms for ns-3. In *Proceedings of the 2015 Workshop on ns-3*, pages 112–119. ACM, 2015.

[5] Sally Floyd, Andrei Gurtov, and Tom Henderson. The newreno modification to tcp's fast recovery algorithm. *RFC2582*, 2004.

[6] Siddharth Gangadhar, Truc Anh N Nguyen, Greeshma Umapathi, and James PG Sterbenz. Tcp westwood (+) protocol implementation in ns-3. In *Proceedings of the 6th International ICST Conference on Simulation Tools and Techniques*, pages 167–175. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2013.

[7] Luigi A Grieco and Saverio Mascolo. Performance evaluation and comparison of westwood+, new reno, and vegas tcp congestion control. *ACM SIGCOMM Computer Communication Review*, 34(2):25–38, 2004.

[8] Tom Henderson, Sally Floyd, Andrei Gurtov, and Yoshifumi Nishida. The newreno modification to tcp?s fast recovery algorithm. *RFC6582*, 2012.

[9] Saverio Mascolo, Claudio Casetti, Mario Gerla, Medy Y Sanadidi, and Ren Wang. Tcp westwood: Bandwidth estimation for enhanced transport over wireless links. In *Proceedings of the 7th annual international conference on Mobile computing and networking*, pages 287–297. ACM, 2001.

[10] NS-3. Ns-3 version 3.24 tcp implementation: http://code.nsnam.org/ns-3.24/file/e8634b0101f7/src/internet/model.

[11] NS-3. Ns-3 version 3.25 tcp implementation: http://code.nsnam.org/ns-3.25/file/3316e06767e7/src/internet/model.

[12] Natale Patriciello and Tom Henderson. Ns-3 version 3.25 tcp implementation: https://www.nsnam.org/bugzilla/show_bug.cgi?id=2188.

[13] Pasi Sarolahti and Alexey Kuznetsov. Congestion control in linux tcp. In *USENIX Annual Technical Conference, FREENIX Track*, pages 49–62, 2002.

[14] Andrew G Shewmaker, Carlos Maltzahn, Katia Obraczka, and Scott Brandt. Tcp inigo: Ambidextrous congestion control. *Technical Report*, 2015.

[15] David X Wei and Pei Cao. Ns-2 tcp-linux: an ns-2 tcp implementation with congestion control algorithms from linux. In *Proceeding from the 2006 workshop on ns-2: the IP network simulator*, page 9. ACM, 2006.