Towards a Modular and Extensible TCP Congestion Control Implementation for the *ns-3* Network Simulator

Allison Hume
University of California, Santa Cruz
Email: ahume128@gmail.com

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

***Abstract*—In contrast to live hardware testbeds, which are largely used to measure performance, network simulators are primarily used to test and validate new protocols and algorithms. As such, a network simulator should be designed to be modular, extensible, re-usable, and robust. This work examines the implementation of TCP congestion control in the *ns-3* network simulator and presents a new design, not specific to *ns-3*, that is more modular and extensible. We also show that the implementation in the current *ns-3* release (version 3.25) is not able to duplicate results presented in previous literature. These findings, along with the improved design, are the main contributions of this paper. To validate our design, we show how its modular approach can be used to implement existing TCP congestion control variants, e.g., TCP New Reno and TCP Westwood.**

I. Introduction and Motivation

Network simulators are an essential component of the network protocol and application development cycle. They complement live hardware testbeds as they enable experimentation under a wide variety of networking environments and conditions. They also allow reproduction of experiments, which is a fundamental scientific methodology principle. Because network simulators are used primarily for testing and validation, their main goal need not be performance but rather to provide a robust environment that can be easily modified and extended so that new protocols can be added to the existing code base with minimal effort. As such, we believe that modularity, extensibility, re-usability, and robustness should be the main focus of network simulator code design.

One active area of research in the networking community is introducing and comparing different TCP congestion control algorithms. Network simulation platforms have played a crucial role in advancing TCP congestion control state-of-the-art and *ns-3*, as well as its previous incarnation, *ns-2*, have a long track record as essential tools for TCP congestion control research [1], [6]. Like any software system, *ns-3* has been evolving, but there has not been enough of an effort to continue to validate and understand the simulator itself.

The work in this paper was motivated by our experience in working on a new variant of TCP congestion control, TCP Inigo [14]. We were planning to use *ns-3* as the simulation platform to validate TCP Inigo and to complement experiments run on a testbed. 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, was not robust enough to support the detailed analysis that is necessary to understand and compare congestion control algorithms.

To illustrate the previous claim, Figure 1 shows a trace of the congestion window in bytes (see Section II for a review of congestion control in TCP) for TCP Westwood [7] using an experiment found in Gangadhar et al [5]. The left plot shows the experiment run on *ns-3* release 3.24 while the plot on the right shows the same experiment run on *ns-3* release 3.25. These results show that the implementation of TCP Westwood has changed dramatically between the two implementations. How can researchers trust simulators to compare new versions of TCP when established algorithms are not consistent between releases? The work we present in this paper has been motivated by these findings and by the difficulties encountered in adding a new congestion control algorithm to *ns-3*.

The main contribution of this work is a new design of the TCP congestion control implementation for network simulators that targets modularity, extensibility, re-usability, and robustness. The design will be presented after a brief description of the history of TCP code design in the *ns-3* simulator. We also present results that question the existing TCP implementation in *ns-3*, lending weight to the argument that a comprehensive design evaluation as well as a formal validation process are very much needed.

II. History of TCP Design in *ns-3*

The previous discussion helps to illuminate some of the challenges that arise when attempting to implement TCP congestion control in a way that is both efficient and extensible. In particular, extensibility in a simulation platform is quite critical as new versions of TCP congestion control have been frequently proposed and added to network simulators for validation and testing. If the TCP implementation in Linux favors efficiency over extensibility that may be acceptable since new TCP variants are not added to the Linux kernel very often. In a network simulator, however, lack of extensibility goes directly against the goals of the software. This section discusses the recent history of TCP congestion control design choices in the *ns-3* network simulator [9], which motivate our new design as described in Section IV. We should point out that even though our work was motivated by *ns-3*'s TCP congestion control implementation, our design is applicable to network simulation platforms in general.

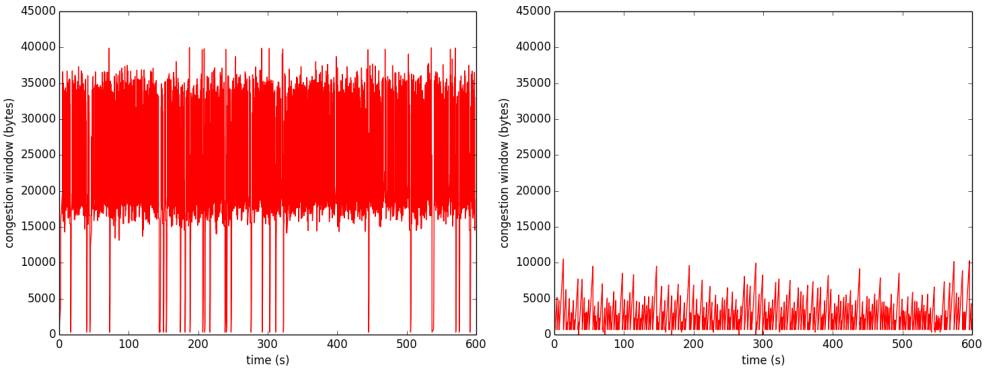

Fig. 1. A 600 second trace of the congestion window (in bytes) of TCP Westwood. Experiment from Gangadhar et al [5] run on *ns-3* release version 3.24 (left) and version 3.25 (right).

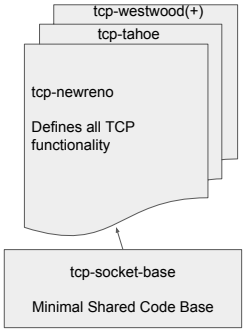

Fig. 2. Block diagram showing the implementation of TCP congestion control for *ns-3* version 3.24.

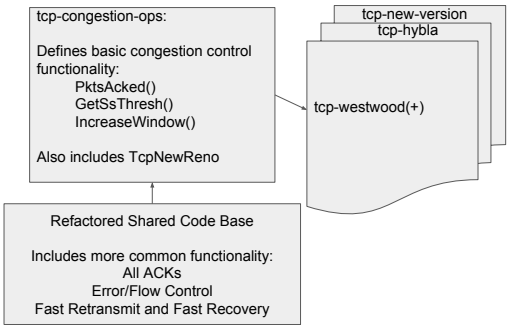

Fig. 3. Block diagram showing the implementation of TCP congestion control for *ns-3* version 3.25.

As shown in Figure 2, the second most recent *ns-3* release, version 3.24, is organized as follows: (1) it has a module called `tcp-socket-base` that contains the code base shared by all versions of TCP implemented in the simulator, and (2) there is one main module for each TCP variant containing most of the code required for that specific congestion control algorithm [10]. The `tcp-socket-base` module contains very little code. It has a function named `ReceivedAck`, called upon the receipt of a new acknowledgment, which serves mostly to call the `NewAck` and `DupAck` functions, which are defined for each TCP variant in its respective module. `NewAck` and `DupAck` do not just contain congestion control functionality, but include all TCP functionality that should be invoked upon receipt of a new or duplicate acknowledgment, respectively. Consequently, although the code is organized by congestion control algorithm, a significant portion of code in each module is in no way specific to the corresponding congestion control algorithm. The main downsides of this design are the significant code duplication and the high overhead required to maintain and extend it, e.g., add a new congestion control algorithm.

Recently, *ns*-3's TCP implementation has undergone a refactoring effort. As shown in Figure 3, in the refactored code the majority of the TCP code is shared and only specific congestion control functions need to 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 shared functionality and calls more specific, well-defined congestion control functions such as `PktsAcked`, `GetSsThresh`, and `IncreaseWindow`. These functions are defined in the `congestion-ops` module. TCP congestion control variants are each implemented in a separate module using the `congestion-ops` module as their base class. The implementation for TCP New Reno, however, exists in `congestion-ops` because New Reno is the default TCP congestion control algorithm used by *ns-3*.

The refactored design in *ns-3* version 3.25 is more extensible than the previous design, although great care must be taken to ensure that all shared code is truly generic. As will be discussed further in Section III, that is not currently the case. Another criticism of this design is that, since TCP New Reno is used as the default implementation, the implementation of other congestion control algorithms is not always completely clear. It is not obvious, for example, that the TCP Westwood implementation actually uses the `SlowStart` function defined for New Reno, since `SlowStart` is not one of the three functions specified in the base class. We argue that, although the current design is moving in the correct direction, additional improvements are needed.

TABLE I
SUMMARY OF SIMULATION PARAMETERS AND THEIR VALUES

Original Source	Simulator Used	Figure	Bottleneck Link	Access Link	MTU Size	Loss Rate
Gangadhar et al [5]	*ns-3*: 3.15 or 3.18 ?	1	2 Mbps, 0.01ms delay	10 Mbps, 45ms delay	400 B	$5 * 10^{-3}$
tcp-variants-comparison	N/A	5				0, 10^{-3}
Casoni et al [3]	*ns-3*: 3.22	*	10 Mbps, 25ms delay	100Mbps, 1ns delay	1500 B	10^{-3}
Grieco et al [6]	*ns-2*	4	2Mbps, 125ms delay	1Gbps, 0.01ms delay	1500 B	0, 10^{-3}

III. VALIDATING *ns-3* VERSION 3.25

Before presenting our proposed design, we further motivate the need for a new design by presenting experiments we conducted to validate the current state of the simulator. To this end, we tried to reproduce results reported in the literature. In particular, the experiments presented here are taken from previous work by Gangadhar et al [5], Casoni et al [3], and Grieco et al [6]. The simulation parameters and the values we used for these experiments are summarized in Table I, which also includes the reference to the work where the experiment was originally carried out ("Original Source" column) and the network simulator used to run the original experiments ("Simulator Used" column). A "?" in that column indicates that the simulator version was not specified in the original paper, so we used the publication date and the simulator's version release schedule to infer the version used in the original experiments. The "tcp-variants-comparison" row in Table I refers to a version of the experiment developed by Gangadhar et al that is provided as part of *ns-3* release (in the examples/tcp directory). Column "Figure" in Table I specifies the figure where the results of the experiments are plotted. A "*" in that column indicates that the figure is not shown due to space limitations.

All experiments involve a source and a sink connected by a gateway with point-to-point links, and traffic sent from the source to the sink using the `BulkSendHelper` traffic generator, which continuously sends traffic to fill the network. The link with the lower bandwidth is referred to as "Bottleneck Link", and the other as "Access Link". Their bandwidth and propagation delay are shown in the fourth and fifth columns of Table I. Where applicable, packet losses are simulated using a uniform distribution defined by the rate shown in the Loss Rate column in Table I.

Different experiments show different degrees of variation between the current *ns-3* 3.25 release and original results obtained from either previous versions of *ns-3*, *ns-2*, or hardware. One experiment is originally from Casoni et al [3], in which results from *ns-3* version 3.22's TCP New Reno were compared to Linux. Their results showed that `cwnd` and `ssth` from *ns-3* were both very close in scale and pattern to the values obtained from Linux; however, `cwnd` showed unexpected spikes in *ns-3* that were attributed to fast retransmit. When repeated on *ns-3* 3.24 we were able to reproduce those results. We found, however, that the spikes are due to fast recovery: 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 [4]). These results are not presented here due to space limitations.

This behavior, however, does not necessarily constitute an error. The *ns-3* code very closely follows the RFC while the Linux TCP implementation handles fast recovery differently. The Linux implementation does not follow RFC 2582 [4] exactly and does not adjust `cwnd` for every ACK during fast recovery [3] [13].

In order to confirm our theory, we wished to disable fast recovery. To do so, however, we had to track down and comment out the relevant code within the shared code base. The current design of TCP in *ns-3* allows no option for changing or disabling the fast retransmit and fast recovery algorithms. In different scenarios you may want different implementations of the algorithm, or, as in this case, simply to turn one off. Commenting out sections of code is a very error prone method of achieving that result.

Although the fast recovery algorithm for NewReno is correct, the same algorithm is applied to TCP Westwood, which is not correct. The Westwood algorithm specifies that the bandwidth estimation should be applied not just after a timeout, but after n duplicate ACKs as well [7]. The New Reno fast recovery algorithm specifies that after three duplicate ACKs the congestion window is halved. Since this New Reno code exists in the shared code base, after three duplicate ACKs the congestion window is halved for TCP Westwood as well, and the bandwidth estimation does not occur as it should, according to the Westwood specification. Having one fast retransmit and fast recovery implementation in the shared code base is therefore not a viable design for an extensible TCP implementation as it is not guaranteed that any two congestion control algorithms will require the same behavior for those components.

We also found, upon commenting out the fast recovery implementations, that the new version of *ns-3* responds differently to the removal of the algorithm than did *ns-3* 3.24. Specifically, slow start is initiated an order of 10 times more frequently. Further discussion of this issue is outside the scope of this paper, but it is clear that the current state of *ns-3* is not sufficiently validated.

Figure 4 shows the results of an experiment comparing *ns-3* versions 3.24 and 3.25. This experiment was first presented in Grieco et al [6] and seen also in Abdeljaouad et al [1]. It originally included one forward flow with 10 back-flows turning on and off but, due to the results obtained while attempting duplication, the experiment is presented here without

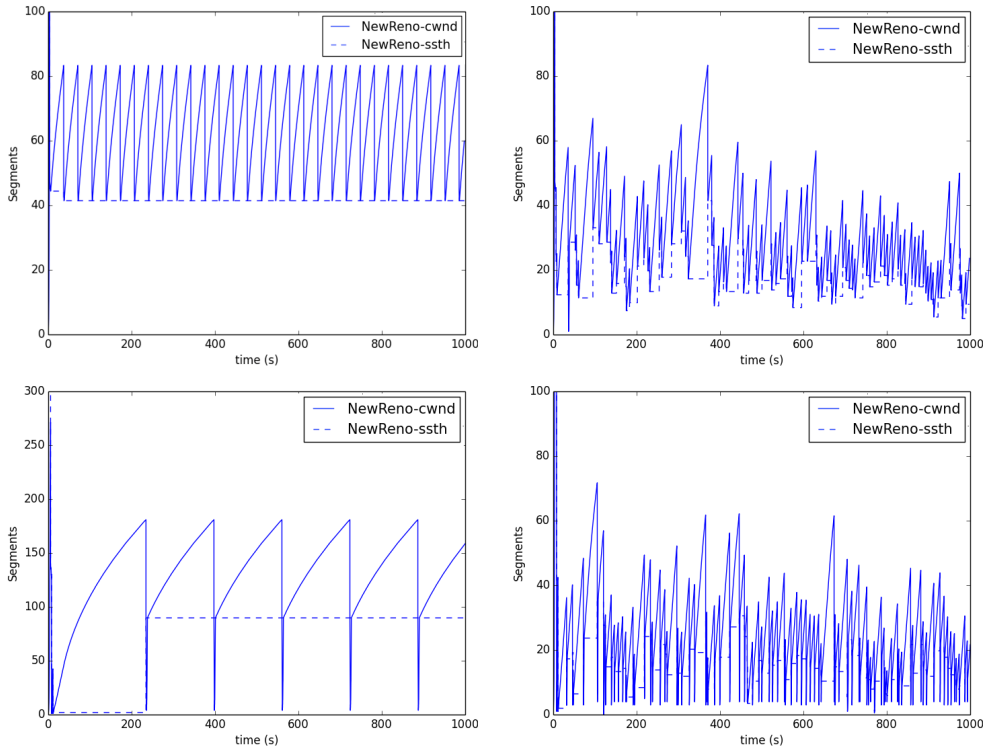

Fig. 4. Congestion window and slow start threshold for New Reno in *ns-3 3.24* (top row) and 3.25 (bottom row), with packet loss rates set to 0 (left column) and 10^{-3} (right column). Full trace from a 1000 second experiment.

the back-flows. The top row of the figure shows results for version 3.24 and the bottom row for version 3.25. The left- and right columns show results using packet loss rates of 0 and 10^{-3}, respectively.

The differences in the plots with an error rate of 0 (left column) are quite significant. The top left plot very closely matches results previously obtained by Grieco et al [6] and Abdeljaouad et al [1], which used the *ns-2* network simulator [8]. It appears, looking at the bottom left plot, that in *ns-3* version 3.25, without packet loss, congestion events only occur once the congestion window has grown to a much larger size than in *ns-3* version 3.24. Since the experiments have the exact same setup, it is clear that some parameter is not enforced properly in version 3.25. This is not visible in the equivalent experiments with a non-zero packet loss rate (right column) because even a very small loss rate creates enough perceived congestion events to mask this issue. The difference in these plots speaks to an insufficient validation effort between release versions of *ns-3*. If such simple experiments are not repeatable, how can the implementation be used to support further research?

Figure 5 shows behavior similar to the plots in Figure 4. The plots in this figure were generated using the `tcp-variants-comparison` experiment that exists in the examples/tcp directory in every *ns-3* release (based on the experiment presented in Gangadhar et al [5]). Similarly to Figure 4, the top row shows experiments run on release 3.24, the bottom row experiments run on release 3.25, the left column plots packet loss rate of 0, and the right column loss rate of 10^{-3}. Again, the difference in the shape of `cwnd` for 0 loss rate indicates lack of repeatability even for an example provided as part of each *ns-3* release.

IV. Proposed Design

The examples presented in the previous section motivate the need for re-designing a TCP congestion control implementation for network simulators in order to increase modularity, and consequently robustness and extensibility. Our design is based on the following goals:

Extensibility: The primary goal of network simulators is to allow researchers to test and validate new protocols or variants/extensions to existing ones. TCP is a prime example: as one of the most widely used Internet protocols, TCP has attracted considerable attention from the networking research and practitioner communities. In particular, the TCP congestion control algorithms have evolved significantly over the years, with new versions often appearing as minor modifications of existing algorithms. Consequently, it must be straightforward to add new TCP congestion control variants to network simulators' existing TCP code base in order to facilitate testing and validation.

Modularity: TCP congestion control has evolved significantly over time and currently has many variants, each of which consists of a set of algorithms. Among those algorithms, some may be unique and some shared with other variants. As

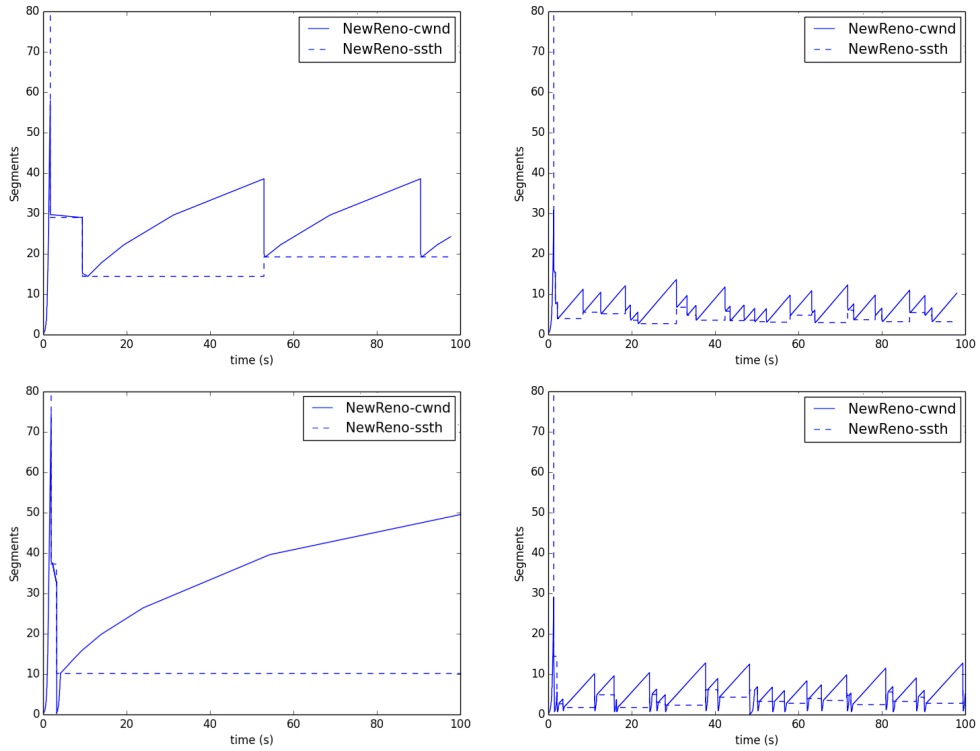

Fig. 5. Congestion window and slow start threshold for New Reno in *ns-3 3.24* (top row) and 3.25 (bottom row), with packet loss rates set to 0 (left column) and 10^{-3} (right column). Full trace from a 100 second experiment.

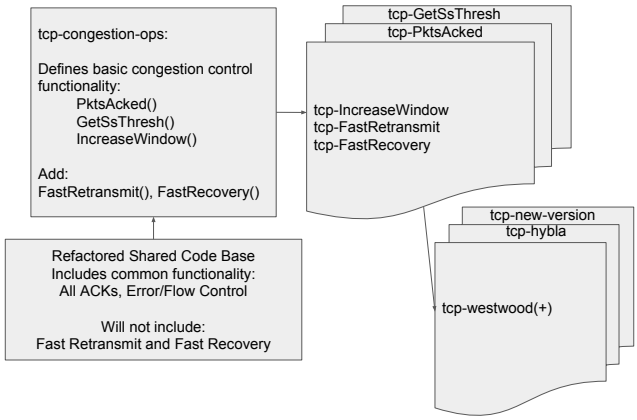

Fig. 6. Proposed design for a modular and extensible TCP congestion control implementation.

such, modular design and implementation are critical, especially in network simulation platforms. Modularity will allow different congestion control variants to share implementations of common functions that have been well tested. It will also facilitate experimenting and testing new congestion control versions since modules that implement different functions can be easily enabled/disabled. In addition, it will greatly reduce the overhead necessary for implementing new congestion control mechanisms, as they are often "small" modifications to one component of existing algorithms.

While our design can be generalized to any network simulator, we used *ns-3* to showcase and test our implementation. As such, we will continue to use *ns-3*'s TCP congestion control implementation as the basis for our design changes.

The main modification to the shared code found in `tcp-socket-base` is that it no longer includes fast retransmit and fast recovery algorithms. The `tcp-congestion-ops` module remains the base class for all congestion control implementations with the following changes: (1) it no longer contains the implementation of New Reno, which now is implemented in a separate, dedicated module, and (2) it now includes `FastRetransmit` and `FastRecovery` function signatures. Each function in the base class will have a corresponding module in which all its variant implementations will reside. Lastly, each specific congestion control algorithm will have a module that defines the selection of its components.

Figure 6 summarizes the proposed design. We believe that this design achieves modularity and extensibility, ensuring implementations are clear and easy to understand. Specifically, having a module for each algorithm, not just each congestion control variant, allows developers to easily find and follow implementations that already exist for one specific piece of functionality, without having to root through all congestion control modules and the shared code base.

Table II summarizes the changes made between *ns-3* releases 3.24 and 3.25, as well as our design changes. Each

TABLE II
SUMMARY OF DESIGN CHANGES

Design	Shared Code Base	Congestion Module	Component Modules	Algorithm Modules
***ns-3* 3.24**	Minimal	N/A	N/A	Whole implementations
***ns-3* 3.25**	ACKs, Error/Flow Control, Fast Retransmit and Fast Recovery	Defines congestion control as: PktsAcked, GetSsThresh, IncreaseWindow. Contains NewReno implementation	N/A	Unique components of implementation
Proposed	Remove Fast Retransmit and Fast Recovery	Add FastRetransmit and FastRecovery to congestion control definition. Remove NewReno to algorithm module	Add module for each of: IncreaseWindow, GetSsThresh, PktsAcked, FastRetransmit, FastRecovery	Uses components defined in component modules

column in the table corresponds to a specific module and the rows describe the state of that module for the three designs. The table clearly shows the proposed design's increased modularity. The extensibility of the proposed design can be inferred from the ease with which a new algorithm module can be created, given an existing set of component modules.

V. DESIGN VALIDATION

A. TCP New Reno

Ideally we would like to fully validate our proposed design. Unfortunately, as demonstrated previously, the starting point for our design is not entirely correct. In this section we show that we are able to obtain the same results for TCP New Reno using our design and implementation when compared to the results obtained with *ns-3 3.25*. The second part of this section shows the same results for TCP Westwood and contains a deep dive into an example illustrating our attempts at achieving consistent results between *ns-3* 3.24 and 3.25. The goal is to show how high a barrier poor validation between releases can be for researchers trying to validate their results.

The plot on the left side of Figure 7 shows results obtained by running the experiment by Casoni et al [3] using our implementation of TCP New Reno, and matches the previous results obtained with that experiment. The plot on the right side of Figure 7 shows results from the `tcp-variants-comparison` experiment available as part as the *ns-3* 3.25 release. It matches quite well Figure 5's lower left graph which was obtained by running the same experiment using *ns-3* 3.25's TCP NewReno.

B. TCP Westwood

For TCP Westwood, we used the experiment from Gangadhar et al [5]. Figure 1 shows the original outcome of this experiment, in which we observe considerable differences between *ns-3* releases 3.24 and 3.25. As discussed in Section III, one clear bug with the TCP Westwood implementation in release 3.25 is that it uses New Reno's implementation of fast recovery, instead of the bandwidth estimate. We hoped that fixing this problem with the TCP Westwood implementation, which was done during the implementation of the proposed design, would yield behavior similar to release 3.24. The results, however, continued to look almost identical to those shown in the second plot of Figure 1.

Although this confirms that our design does not add additional errors, it did not enable us to fully understand the Westwood implementation in *ns-3* 3.25. This prompted a deeper investigation into the root cause of these differences because, in contrast to the differences seen in TCP NewReno, the differences in TCP Westwood are on a large enough scale to be deeply concerning. This investigation is outlined here to underscore the need for careful and comprehensive validation between simulation releases.

In addition to the problem of using New Reno's fast recovery, instead of TCP Westwood's original bandwidth estimator, our investigation uncovered that in *ns-3* 3.25's TCP Westwood implementation, `ssth` was not being updated correctly. The following trace shows the first few values of `ssth` recorded in the simulation for *ns-3* 3.24 ("..." indicates value was unchanged for several time steps):

```
time ssth
0.0 65535
0.0905768 262140
...
0.5562 20225
```

And the following is the equivalent trace for *ns-3* 3.25:

```
time ssth
0.0 0
0.0905768 4294967295
...
0.5562 680
```

The first obvious difference is that the initial `ssth` value has changed. It is not clear why this initial value was changed to 0 as the specification for `ssth` is "the initial value of [ssth] SHOULD be set arbitrarily high" [2]. The next item in the trace shows that there was an overflow event, as `ssth` is an unsigned integer. Finally, the trace shows `ssth` being set to 680 which, for this experiment, is twice the segment size. This is the smallest value to which `ssth` should ever be set, and it should only be set to that value if the bandwidth estimator in Westwood would cause it to be set to a value smaller than that. In version 3.25, for every call of `GetSsThresh`, no value other than 680 is ever returned, meaning that the bandwidth estimator continuously produces very small bandwidth values. For version 3.24 that value is only returned a handful of times.

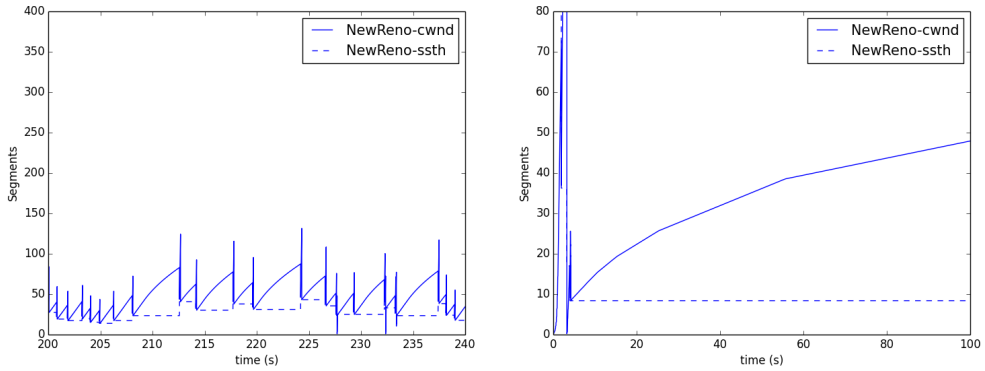

Fig. 7. Comparison between simulation results obtained with our design and implementation of New Reno and *ns-3 3.25*'s New Reno. Right-side graph can be compared with Figure 5's lower left graph.

The bottleneck bandwidth for the experiment is 2 Mbps but the values returned by the bandwidth estimator were on the order of 50 Kbps. The bandwidth estimator is calculated using the following equation in the Westwood implementation: $BW = PktsAcked * SegementSize/RoundTripTime$. `RoundTripTime` is estimated using send and receive time for packets and ACKs, respectively, and was consistently close to the 90ms which was expected for this experiment. One obvious problem is that the segment size is not the full size of the packet, but this would not account for such a large difference between expected and estimated bandwidth. The problem must, therefore, be an issue with the counting of the number of acknowledged packets.

At this point three bugs have been discovered in *ns-3* 3.25's TCP Westwood implementation, namely: (1) incorrect usage of the New Reno fast recovery algorithm, (2) incorrect initialization of `ssth`, and (3) incorrect operation of the bandwidth estimator. To understand if the bandwidth estimator was the major factor contributing to the differences observed in TCP Westwood between the two releases, the result of the bandwidth estimator was multiplied by 40 (the average factor between the expected result and actual result of the estimator). This plot is much closer in scale to the plot on the left of Figure 1, from *ns-3* 3.24, but is still different enough to suggest that there may be additional problems with *ns-3* 3.25's TCP Westwood implemention.

The large number of issues encountered while trying to reproduce such simple experiments demonstrates the need for modular, extensible design as well as comprehensive implementation validation.

VI. Conclusions

This paper presented a modular and extensible TCP congestion control design for network simulators. We showcase our design using the *ns-3* network simulator. In addition, we showed that the recent *ns-3* code refactoring caused errors to be introduced in the TCP implementation and the difficulty that those errors create when trying to work with- and extend the existing TCP implementation. We argued that in addition to a modular and extensible design, a comprehensive validation process for network simulators must be adopted. This is critical in order to support protocol and application developers who rely on simulators to understand, propose, validate, and test their work.

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] 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.

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

[5] 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.

[6] 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.

[7] 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.

[8] NSNAM. The network simulator - ns-2: http://www.isi.edu/nsnam/ns/.

[9] NSNAM. ns-3 https://www.nsnam.org/.

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

[11] NSNAM. 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.