On TCP Simulation Fidelity in ns-2

Lingfeng Guo
Department of Information Engineering
The Chinese University of Hong Kong
Hong Kong
gl016@ie.cuhk.edu.hk

Jack Y. B. Lee
Department of Information Engineering
The Chinese University of Hong Kong
Hong Kong
yblee@ie.cuhk.edu.hk

ABSTRACT

The Network Simulator version 2, also known as ns-2, is a widely used platform for network and protocol performance evaluation. Over the years it has benefited from numerous studies in improving its simulation fidelity. Nevertheless, this study discovered that ns-2's TCP simulation accuracy could be impaired substantially in cases where the first-hop link is the bottleneck. This is common in many applications where the client host uploads data to Internet servers as the uplink, e.g., wireless and mobile networks, may have far lower bandwidth than the Internet core. This work investigated this performance anomaly by dissecting and comparing ns-2's implementation against Linux implementation; and by developing extensions to ns-2 to resolve the anomaly as well as five additional updates to bring its implementation to match recent Linux implementations. Extensive verifications against experiments conducted in a physical testbed confirmed the accuracy of the extended ns-2, offering a renewed and accurate simulator for mobile and wireless networking.

KEYWORDS

ns-2; TCP Congestion Control; Intra-Host-Back-Pressure (IHBP); Buffer Overflow

ACM Reference format:

Lingfeng Guo, Jack Y. B. Lee. 2018. On TCP Simulation Fidelity in ns-2. *In 14th ACM International Symposium on QoS and Security for Wireless and Mobile Networks (Q2SWinet'18).* October 28-November 2, 2018, Montréal, Québec, Canada. ACM, NY, NY, USA. 8 pages.
https://doi.org/10.1145/3267129.3267132

1 INTRODUCTION

The Network Simulator version 2, also known as ns-2 [1], is one of the most widely used network simulators in networking fields. In particular, ns-2 can incorporate part of Linux's TCP congestion control modules into the simulator which enables direct simulation of Linux TCP congestion modules. For this reason a large number of previous works employed ns-2 in the design, optimization, and evaluation of a wide range of TCP congestion control algorithms (e.g., [2-6]). At the time of writing Google scholar returned over ten thousand papers referencing ns-2 which clearly demonstrates its wide impact.

Consequently, the simulation fidelity of ns-2 is of great significance to the research community and a number of previous works [7-8] have improved ns-2's accuracy over the years. Nevertheless, with continuous development and refinement of TCP implementations in production operating systems it is critical to assess ns-2's accuracy against recent TCP implementations.

In this study we first conducted a systematic assessment of ns-2 against the current TCP implementation in Linux. The latter choice is motivated by the wide-spread deployment of Internet services using Linux-based servers. Our investigations revealed that there is a fundamental difference in behavior between the TCP simulated by ns-2 and the one as implemented in Linux despite the fact that ns-2 is already running the Linux TCP congestion control codes directly. This difference, which we termed Intra-Host-Back-Pressure (IHBP), can lead to significant differences in simulated vs real TCP behaviors when the first-hop is the bottleneck link. This is now very common as many mobile applications such as social networks running in smartphones upload data (e.g., photos, videos) to Internet servers.

In addition, a detailed analysis of the ns-2 codes against Linux kernel sources further revealed new features in Linux that are missing in ns-2 and differences in implementation of known features. These differences further impair ns-2's simulation fidelity of TCP even when the bottleneck is not at the first hop.

To tackle these limitations we developed a solution to rectify the IHBP problem and updated the current ns-2 codebase (version 2.35) to bring it in line with recent Linux implementations. We validated the modifications by comparing ns-2 results to a physical testbed with the same topology running Linux. The source codes for the modifications are available online [9].

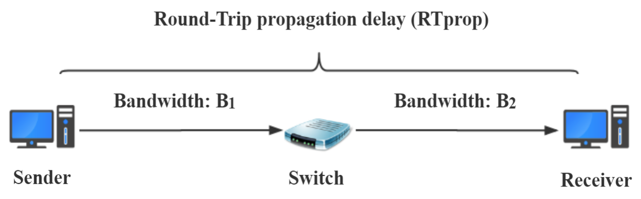

Figure 1: Topology for simulation and testbed.

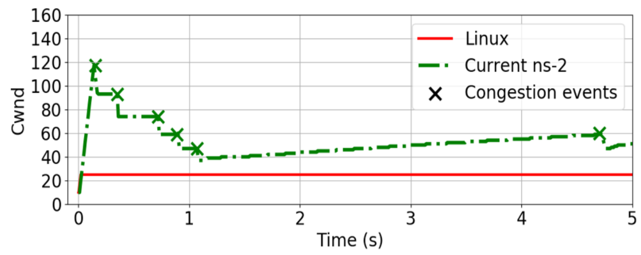

Figure 2: Comparison of cwnd trajectories, showing how current ns-2 deviates from Linux due to unexpected congestion events.

The rest of this paper is arranged as follows: Section 2 reviews the evolution of ns-2; Section 3 reports the performance anomalies of current ns-2 and analyzes their root causes; Section 4 presents the extended ns-2; Section 5 validates the extended ns-2 against Linux; Section 6 summarizes the paper and outline future work.

2 RELATED WORK

Given ns-2's wide-spread use in networking research a number of researchers have investigated and improved its accuracy over the years. An early work by Jansen and McGregor [8] proposed the Network Simulation Cradle to incorporate codes from network stacks as implemented in production operating system into the ns-2 simulator. Their results demonstrated that NSC can substantially improve ns-2's accuracy against real-world TCP implementations.

In another study Wei and Cao [7] developed an ns-2 TCP implementation with congestion control algorithms taken directly from the Linux kernel sources. Their work had been adopted into the official ns-2 codebase since version 2.33 and has been the recommended implementation.

More recently, the next-generation network simulator, also known as ns-3, is being actively developed [10]. ns-3 is a complete redesign and is gaining momentum in the research community. Nevertheless for TCP-related research ns-3 still lacks support for critical components including TCP Cubic [2] (Linux default), TCP Compound [3] (Microsoft Windows default), and the capability to import current TCP modules from Linux into the simulator. Most importantly, a vast body of existing and on-going studies are based on ns-2 and thus it is critical to reexamine ns-2's simulation fidelity. In addition, the findings in this work can also inform the future development of ns-3.

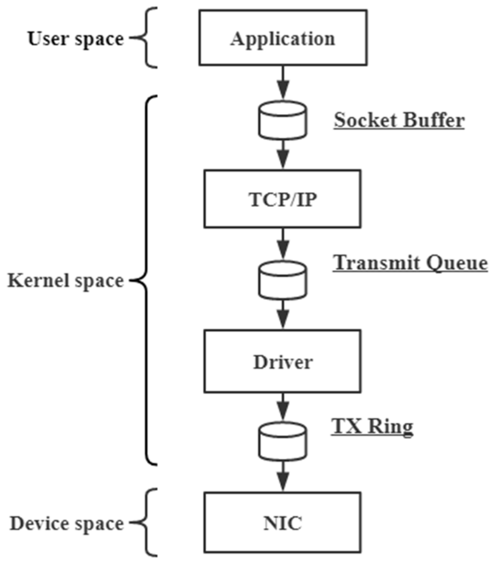

Figure 3: Intra-host buffering between TCP and network interface card (NIC) in Linux.

3 NS-2 TCP SIMULATION REVISITED

In this section we report our key findings on ns-2's accuracy in simulating TCP by comparing the simulated TCP behavior with a physical testbed running Linux (Ubuntu 16 with kernel 4.12.1) hosts connected by a Cisco switch. Both simulation and the testbed employed the same two-hop topology as shown in Fig. 1. The links' bandwidth (B_1, B_2) are configurable between 10Mbps, 100Mbps, and 1Gbps. By configuring different bandwidth for B_1 and B_2 we can position the bottleneck at either the first-hop or the second-hop.

3.1 First-Hop Bottleneck

First, we investigated the case where the first-hop is the bottleneck by configuring the first- and second-hop link bandwidth to B_1=10Mbps and B_2=1Gbps respectively. The round-trip propagation delay (RTprop) is set to 10ms. Note that in ns-2 there is a queue associated with every link. For the first-hop link the queue is interfacing to the sender and we adopted ns-2's default queue size of 50 packets. For the second-hop link we matched its queue size to the buffer size of the testbed switch (Cisco WS-C2960S).

We employed TCP Cubic as the congestion control algorithm in both simulation and experiment. It is worth reiterating that ns-2 directly imports the TCP Cubic module from Linux kernel so the congestion control implementations are exactly the same.

For both simulation and experiment a TCP connection was setup between the sender and receiver, and then the sender continuously transmitted data as fast as TCP allowed. We plot in Fig. 2 the trajectories of the congestion window (cwnd) from ns-2 (TCP parameters were recorded every 10ms) and testbed (via tcpprobe) over time. Surprisingly, the two cwnd trajectories differ significantly early on. In the case of ns-2 its cwnd kept on increasing until it encountered the first packet loss at time 0.16s and continued to encounter packet losses after that. By contrast, shortly after an initial increase Ubuntu settled on a cwnd of 25 and remained unchanged. Unlike ns-2 there was no packet loss during the whole experiment.

Clearly there was something amiss in the ns-2 simulation. As the Cubic congestion module was imported directly from Linux kernel it is unlikely due to discrepancies in the congestion control algorithm. Given that the first-hop link is the bottleneck at 10Mbps, there should *not* be any queuing, let alone packet drops at the switch as the switch's outgoing link is operating at 1Gbps. Indeed, analysis of ns-2's trace data confirmed the lost packets were not dropped at the switch but at the *sender* due to buffer overflow at the input of the first-hop link.

This can occur in ns-2 because there is no *flow control* between the TCP sender inside the sending host and the first-hop link it was directly connected to. Therefore as TCP ramped up its cwnd during slow-start it eventually exceeded the bandwidth limit of the first-hop link. In this case TCP segments will start to queue up at the first-hop link until the queue becomes full and consequently packets dropped.

Linux's implementation [11] as depicted in Fig. 3 does not suffer from the same problem as there is *implicit* flow control between the TCP sender and the lower layers. Briefly speaking, whenever the lower layers' (i.e., IP, network card driver) buffers are full Linux will *block* TCP from submitting additional segments to IP for transmission. This *intra-host-back-pressure* forces TCP to buffer data internally (e.g., via sockets buffer) and in case the internal buffer is full the application itself will be blocked from sending more data (e.g., via blocking the send() function in the sockets API).

A commonly-employed remedy to the above problem is to configure a very large queue size for the first-hop link to reduce the likelihood of buffer overflow. However this creates another problem – increased delay. Specifically, a TCP sender in both ns-2 and Linux treats a segment submitted to the lower layer as *transmitted* and as such, marks the packet's transmission timestamp at that instant. Thus any queueing time experienced by the TCP segment *inside* the host will count towards the packet's RTT when the corresponding ACK returns. As TCP relies on packet RTT measurements for timeout/retransmission this could lead to inaccurate TCP behavior.

To demonstrate this problem we conducted another ns-2 experiment by configuring the first-hop link buffer to a very large value (100M packets) so that sender buffer overflow was eliminated. Fig. 4 plots the cwnd evolution and the RTT for each packet under ns-2 and Linux respectively. The large first-hop buffer resulted in even larger differences in the cwnd trajectory and the extra queueing resulted in increased RTT as measured by the TCP sender. These are clearly inaccurate simulation of actual TCP behavior.

First-hop bottleneck exists in many real-world networks, most notably client-side data upload scenarios. In these cases, e.g., ADSL, mobile network, etc., the first-hop uplink is often the bottleneck link. Therefore ns-2's simulation fidelity in these network scenarios is of critical importance.

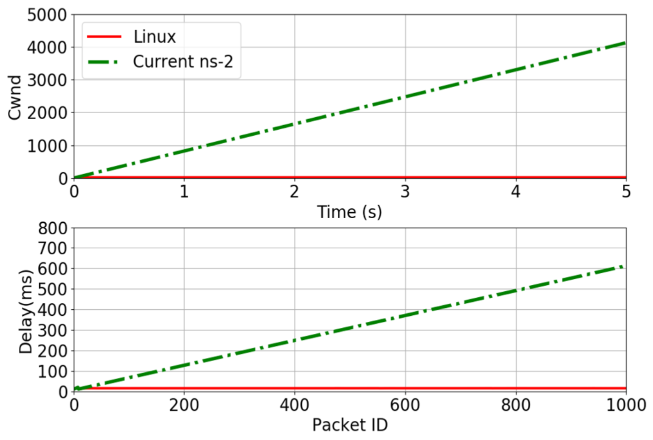

Figure 4: Configuring a large buffer for the first-hop link can prevent buffer overflow but will result in significantly increased packet delay.

3.2 Recent Linux TCP Developments

The TCP module in ns-2 was last updated in 2011. Since then the TCP implementation in Linux has undergone several updates which affected TCP's behavior under certain conditions. Through analyzing and comparing the source codes of ns-2.35 and Linux kernel 4.12.1 we discovered five key updates which can result in different TCP behavior between the two platforms. We summarize our findings below.

Hystart – During slow-start TCP begins with a small cwnd and then increases it rapidly in an exponential manner. In some cases (e.g., large buffer size at the bottleneck [12]) this may result in a sufficiently large cwnd that can exceed the bottleneck bandwidth, leading to packet losses. To tackle this problem Ha and Rhee proposed Hystart [12] to dynamically adapt the exit point for the slow-start phase. Since kernel 2.6.29 Hystart has been adopted in Linux but has not yet been incorporated into ns-2.

Cwnd deduction – [2] suggested to deduct Cubic's cwnd by (1-819/1024) or ~20% upon detecting a congestion event. This has been adopted in ns-2. However, in recent Linux kernel the amount of deduction has been changed to (1-717/1024) or ~30%.

Delayed-ACK – Related to Hystart our study also revealed another subtle issue – delayed ACK [13], which is supported in both current ns-2 and Linux. However, while ns-2 performs delayed-ACK from the beginning of a TCP connection, Linux's implementation is more complicated. Specifically, Linux suppresses delayed-ACK at the beginning of a TCP connection until a certain number (e.g., 70) of ACK packets have been generated. We discovered that this difference could lead to differences in the cwnd trajectories under certain network conditions (c.f. Section 5.2).

Cwnd vs packets-inflight – According to RFC2861 [14] TCP's cwnd should not be raised unless it has been fully utilized, i.e., when TCP receives an ACK packet, it first checks whether the number of packets inflight is equal to the current cwnd. If so then TCP will increase cwnd according to the specific congestion control algorithm employed. Otherwise TCP will leave cwnd unchanged as the latter has not yet been exhausted. By contrast, Linux adopts a slightly different mechanism where the threshold is only half of the current cwnd, i.e., as long as half or more of the cwnd is in use TCP will increase cwnd as determined by its congestion control algorithm.

Application-limited TCP sender – In TCP Cubic its cwnd is a cubic function of *time* since the last congestion event. This mechanism plays an important role in maintaining fairness among flows with different RTTs. However, this may also result in unexpected side-effect in case the application suspends data transmission for a period of time. Without application data for transmission TCP obviously will not run into any congestion event. However the cubic function will continue to grow during the idle period. If the idle time is sufficiently long then cwnd could grow to a large value so that a large burst of transmission will result when transmission resumes, potentially overflowing the bottleneck link buffer leading to packet losses.

Google recently developed a remedy for this issue [15] by modifying TCP Cubic to detect application-limited phase so that the idle time is deducted from the non-congestion period used in computing the cubic function, thereby preventing large data bursts once transmission resumes. We note that ns-2 normally will not trigger this issue as its TCP sender always has data for transmission so there is no application idle time. However if one modifies the sender, e.g., to simulate explicit application transmission rate control, then the anomaly may still occur.

4 PROPOSED NS-2 EXTENSIONS

In this section we tackle the limitations reported in Section 3 by extending the ns-2 implementation to support implicit intra-host-back-pressure and by updating the ns-2 implementation to match Linux. We first analyze intra-host buffering in current ns-2 and then present our extensions.

4.1 Current ns-2 Implementation

Current ns-2 uses a *node* to represent both an end-host and a switch (or router). However, buffer in ns-2 is associated with a *link* instead of a node. This is a suitable abstraction for network switches and routers as buffers are typically associated with switch/router ports which are abstracted as links. Inside the sending node the first-hop link's buffer then effectively becomes the intra-host buffer as packets generated by TCP all pass through it before reaching the first-hop link.

We analyzed ns-2's implementation and summarized the logic of two main functions related to packet transmission, namely Send_much() and Recv(), using pseudocode in Fig. 5. First, an ACK reception event results in a call to Recv() which in turn calls Send_much(). The latter continuously sends packets in a loop until cwnd is exhausted. The root cause of the issue here is that Send_much() does not consider if the link has buffer space to accommodate the incoming packets and thus may lead to the intra-host packet drops as reported in Section 3. We present an extension to correct this behavior in the next section.

Send and **Receive** functions in current ns-2

```
Send_much() /*sends out packet(s) */
Begin:
    Load parameters, including cwnd, sequence number.
    While (packet_in_fligt < cwnd)
        Send a packet
End

Recv(ACK ) /*called upon each ACK reception */
Begin:
    If (ACK is duplicated):
        Do retransmission if necessary
    Else:
        Calculate time-related parameters, e.g., RTT.
    Call congestion modules, e.g., Cubic, Reno.
    Call Send_much()
End
```

Figure 5: Pseudocode for TCP implementation in current ns-2.

Send and **Receive** functions in extended ns-2

```
Send_much()  /*sends out packet(s) */
Begin:
    interval = transmission delay
    Load parameters, including cwnd, sequence number.
    While (packet_in_fligt < cwnd)
        If (link buffer is full):
            Schedule(recv(Empty-ACK), "now + interval")
            break
        Else:
            Send a packet
End

Recv(ACK ) /*called on each ACK reception */
Begin:
    If (ACK is Empty-ACK):
        Send_much()
        Return
    If (ACK is duplicated):
        Do retransmission if necessary
    Else:
        Calculate time-related parameters, e.g., RTT.
    Call congestion modules, like Cubic, Reno.
    Call Send_much()
End
```

Figure 6: Pseudocode for TCP implementation in extended ns-2.

4.2 Intra-Host-Back-Pressure (IHBP)

A naïve approach to simulate IHBP would be to check against buffer availability at the link and exit the packet transmission loop in the Send_much() function in case the link buffer is full. However this approach is also inaccurate.

Table 1: Summary of modifications in extended ns-2.

File name	Modified/added	Purpose
Agent.h	Recv()	Update parameters
Classifier.cc	Recv()	Update parameters
Connector.h	Recv()	Update parameters
Delay.cc	Recv()	Update parameters
Object.h	New class NIC	Add new parameters
Object.cc	Recv()	Update parameters
Queue.cc	Recv()	Update parameters

Tcp-linux.cc	Send_much()
	Recv()
		Fix intra-host buffer overflow

Tcp_cubic.cc	Hystart_reset()
	Hystart_update()
	Bic_acked()
	Bictcp_cwnd_event()
		Implement Hystart and fix application-limited bug

Table 2: Summary of testbed and simulation settings.

Configurations	Testbed	Extended ns-2 simulator
OS	Ubuntu 16	N/A
TCP variant	Cubic	Cubic
Queue discipline	N/A	DropTail (1st hop link)
Host buffer size at first-hop link bandwidth	10Mbps: 6 packets 100Mbps: 12 packets 1Gbps: 20 packets	10Mbps: 6 packets 100Mbps: 12 packets 1Gbps: 20 packets
SACK [17]	Enabled	Enabled
Delayed ACK [13]	Enabled	Enabled
TCP Control Block [18]	Disabled	Not supported
Initial ssthresh	2147483647 packets	2147483647 packets
Initial cwnd [19]	10 packets	10 packets
Switch	Cisco WS-C2960S	N/A
Queue discipline	DropTail	DropTail
Link buffer size	40 packets	40 packets

Recall that packet transmission via Send_much() is triggered *only* by the reception of an ACK packet. Consequently, if transmission is suspended due to full buffer then *no* packet will be transmitted until a new ACK is received, even if link buffer becomes available again *before* then. In contrast, Linux will be able to resume transmission once link buffer space is available again, *without* needing to wait for a new ACK to be received.

There are several ways to implement the correct behavior in ns-2. We chose an approach that minimizes code changes so as to avoid any potential side-effects. This is illustrated in Fig. 6 where the new codes are in italic. Specifically, whenever Send_much() is terminated due to link buffer full we insert a special ACK packet reception event, called Empty-ACK, into the ns-2 scheduler. This special event has a schedule calculated to be the exact time at which the head-of-line packet in the connected link will be completely transmitted, i.e., same time for buffer space to become available again.

The Recv() function is modified to handle this special Empty-ACK packet by calling the Send_much() function to resume submitting TCP packets to the link buffer for transmission. Once Send_much() returns, presumably due to link buffer full again, the Recv() function will then return, bypassing the rest of the codes for actual congestion control. Handling of normal ACK packets are not affected.

4.3 Additional Updates

In addition to intra-host-back-pressure we also implemented the five missing/differing mechanisms in ns-2 as discussed in Section 3.3. We summarize the updates below:

Hystart – Hystart was absent from current ns-2 so we implemented it in accordance to the Linux implementation.

Cwnd Deduction – We updated Cubic's cwnd deduction in ns-2 from 20% to 30% to match Linux's implementation.

Delayed-ACK – We added a new option in ns-2 to configure when delayed-ACK should be activated to match Linux's behavior. For example, in our experiments Linux activated delayed-ACK at the 70th ACK. We note that although this option is an approximation of Linux's delayed-ACK behavior, it was shown to be reasonably accurate (c.f. Section 5.2).

cwnd vs packets-inflight – We updated ns-2's slow-start phase behavior such that cwnd is allowed to grow as long as half of the cwnd has been utilized.

Application-limited TCP sender – We updated the TCP Cubic implementation in ns-2 to exclude application idle time from the non-congestion period used in computing the cubic function.

Together with the IHBP mechanism these were implemented in ns-2 version 2.35 using approximately 200 lines of new codes distributed in nine source files. Table 1 lists the source files updated and summarizes the affected functions.

5 SIMULATOR VERIFICATION

In this section we validate the accuracy of the extended ns-2 simulator by comparing simulated results to experimental results obtained from a physical testbed running Linux hosts connected by an Ethernet switch with equivalent topology and configurations.

5.1 Simulation and Experiment Setup

Both simulation and experiment employed the same network topology as depicted in Fig. 1. Table 2 lists the configurations adopted in ns-2 and testbed. Many of the ns-2 default parameters do not match the testbed so we explicitly configured them to match the testbed parameters

For the host buffer, i.e., the buffer between TCP sender and the network interface card, Linux implements buffering via a mechanism known as TCP Small Queue (TSQ) [16]. This was introduced to improve fairness among competing flows within the same host. We added codes to the Linux kernel to determine the runtime buffer size allocated by TSQ which was shown to be primarily link-bandwidth dependent. We adopted the observed intra-host buffer size in our extended ns-2 simulator.

Current Linux also implements TCP Control Block (TCB) [18] which caches certain TCP parameters such as ssthresh from previous connection for use in new ones. This only affects scenarios with repeated connections from the same peer. TCB is not currently supported by ns-2 and so we disabled the same in the testbed. We are currently investigating the feasibility of implementing TCB in our extended ns-2 simulator.

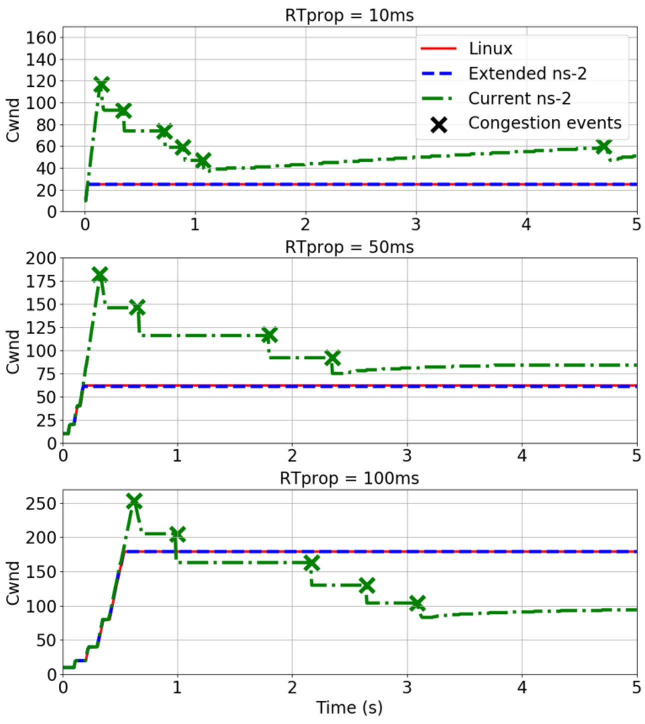

Figure 7: Comparison of cwnd trajectories with first-hop bottleneck of 10Mbps for round-trip propagation delays of 10ms, 50ms, and 100ms.

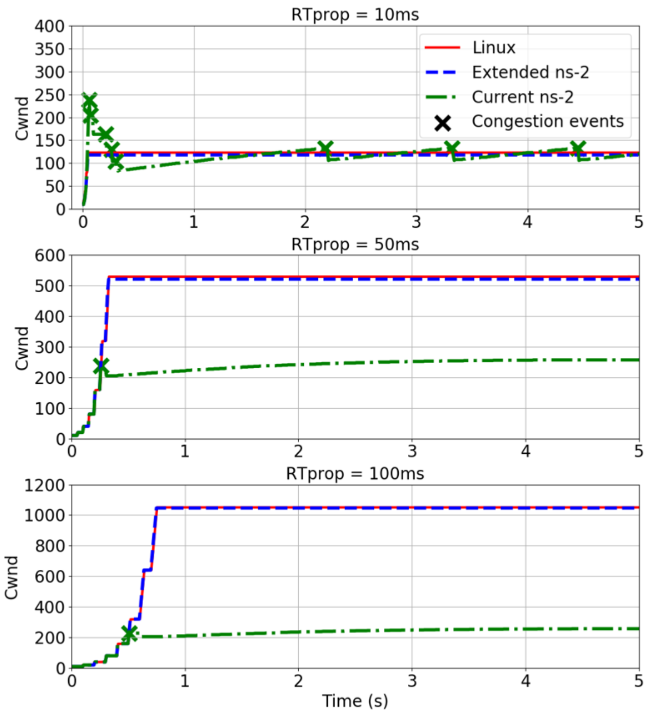

Figure 8: Comparison of cwnd trajectories with first-hop bottleneck of 100Mbps for round-trip propagation delays of 10ms, 50ms, and 100ms.

For the testbed, link bandwidth was controlled by configuring the Ethernet network interface to operate in 10Mbps, 100Mbps, or 1Gbps mode using ethtool [20]. Link delays of {10ms, 50ms, 100ms} were emulated using netem [21]. We employed iperf [22] to generate TCP traffic. Access to internal TCP parameters such as cwnd and ssthresh were obtained via tcpprobe [23]. In all experiments full packet traces were captured using tcpdump [24] for subsequent analysis.

5.2 First-Hop Bottleneck

We first revisit the network topology with the first-hop link being the bottleneck. Fig. 7 and Fig. 8 compare the TCP cwnd trajectories over time for the first-hop link bandwidth of 10Mbps and 100Mbps respectively (the second-hop link bandwidth was fixed to 1Gbps for both cases). Each figure comprises three sub-figures for round-trip propagation delays of 10ms, 50ms, and 100ms respectively. Comparing to the Linux testbed the current ns-2 clearly exhibited very substantial deviations in its cwnd trajectories in all six tested cases.

In the ns-2 simulations there were packet losses caused by buffer overflow at the first-hop link, resulting in TCP's cwnd deduction. As discussed in Section 3 this will never occur in Linux as its intra-host-back-pressure mechanism will prevent TCP from sending more packets than can be buffered. This is verified by Linux's cwnd trajectories which have no packet loss. Most importantly, the cwnd trajectories of the extended ns-2 tracked the Linux ones remarkably closely, exhibited no packet loss as expected.

We note that in addition to implementing the IHBP mechanism the five updates in Section 4.3 also made contribution to the accurate simulation results in the extended ns-2. Due to space limitation these additional verification results were omitted here.

Next we turn our attention to throughput performance in Fig. 9 and Fig. 10. In Fig. 9 where the first-hop link bandwidth was 10 Mbps, the long-term throughputs in the stable region of current ns-2, extended ns-2, and Linux are similar. However current ns-2 exhibited throughput drops as if it encountered congestions. This is clearly not congestion-related as the first-hop was the bottleneck and is a direct consequence of intra-host buffer overflow when the cwnd grew beyond the first-hop link's capacity.

Finally we compare the packet-level delays in Table 3. For each configuration the packet delay was computed from averaging the RTTs of all packets. In the testbed this was done by analyzing the packet trace captured by tcpdump in the sender.

Comparing the first set of results for the 10Mbps bottleneck link case it is clear that current ns-2 exhibited significantly longer delays. This is due to queueing inside the intra-host buffer which, as discussed in Section 3.1, was counted by TCP as part of the RTT. By contrast, the extended ns-2 exhibited similar delays to the testbed. The minor differences are likely due to processing delays in the testbed.

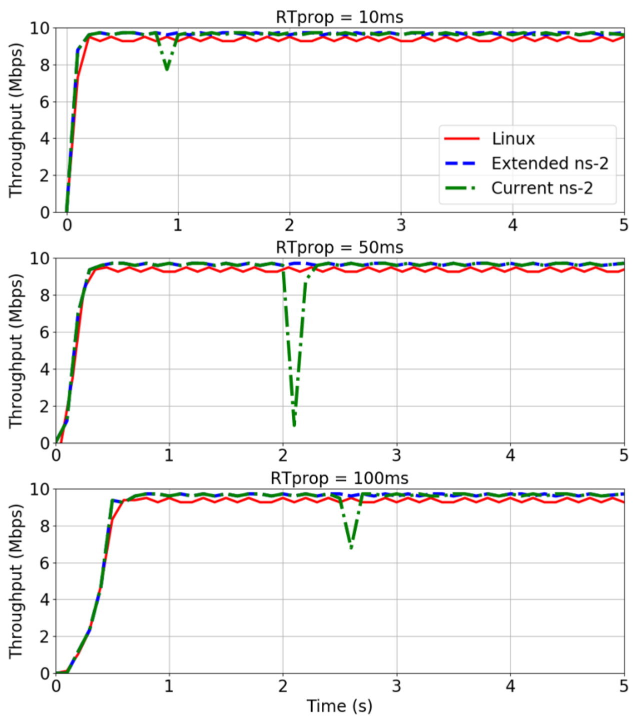

Figure 9: Comparison of throughput trajectories with first-hop bottleneck of 10Mbps for round-trip propagation delays of 10ms, 50ms, and 100ms.

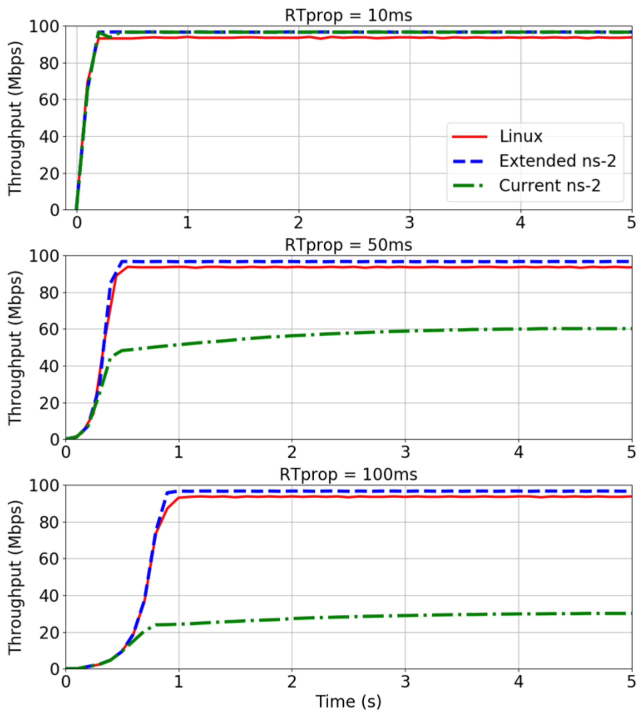

Figure 10: Comparison of throughput trajectories with first-hop bottleneck of 100Mbps for round-trip propagation delays of 10ms, 50ms, and 100ms.

Table 3: Comparison of average packet delay (in ms).

RTprop	Extended ns-2	Current ns-2 (Default buffer)	Current ns-2 (Large buffer)	Linux Testbed
		First-hop bottleneck link at 10Mbps		
10ms	17	59	485	18
50ms	57	92	595	58
100ms	107	134	636	108
		First-hop bottleneck link at 100Mbps		
10ms	11	14	746	12
50ms	51	50	737	52
100ms	101	100	663	101

Interestingly, current ns-2 apparently did not exhibit delay anomalies in the {100Mbps, 50ms} and {100Mbps, 100ms} cases. This is because early packet losses due to intra-host buffer overflow caused TCP Cubic to enter the congestion avoidance phase during which the cwnd was only increased slowly (c.f. Fig. 8). As a result the throughput was in fact *below* the link bandwidth of the first-hop (c.f. Fig. 10) and hence there was little to no queueing inside the host to increase packet delay.

Finally, a known trick to prevent intra-host buffer overflow is to increase the first-hop link buffer size to a large value (e.g., 100M packets). We tested this method and while it can indeed prevent buffer overflow, packets queueing inside the large buffer will experience extended delays as listed in Table 3. Obviously such large delays are far from the actual TCP behavior.

5.3 Non-First-Hop Bottleneck

In this section we relocate the bottleneck from the first-hop link to the second-hop link by setting the former to 1Gbps and the latter to 10/100Mbps. There are two goals: (a) to validate the extended ns-2's behavior against real Linux to ensure that the extensions do not create unintended side-effects; and (b) to demonstrate the added fidelity due to the updates described in Section 3.2.

First, we compare in Fig. 11 the cwnd trajectories of Linux, current and extended ns-2 for the case of 10Mbps bottleneck at the second-hop link. Although intra-host buffer overflow did not occur in this topology, current ns-2's cwnd trajectory still deviated significantly from Linux at the beginning of the connection. This is due to Linux's Hystart [12] mechanism which is absent from current ns-2. It prevented Linux TCP from growing its cwnd too aggressively at the beginning which could result in early congestion in low-bandwidth links as demonstrated by the extra congestion events early on in current ns-2's cwnd trajectories. By contrast, with Hystart incorporated the extended ns-2 closely tracked Linux's cwnd trajectories.

Fig. 12 plots the cwnd trajectories for the case of 100Mbps bottleneck at the second-hop link. With the higher link bandwidth the impact of Hystart became smaller. Nevertheless current ns-2 still exhibited some deviations from Linux especially at longer propagation delays of 50ms and 100ms. By contrast, the extended ns-2 tracked Linux's cwnd trajectories consistently, demonstrating its simulation fidelity.

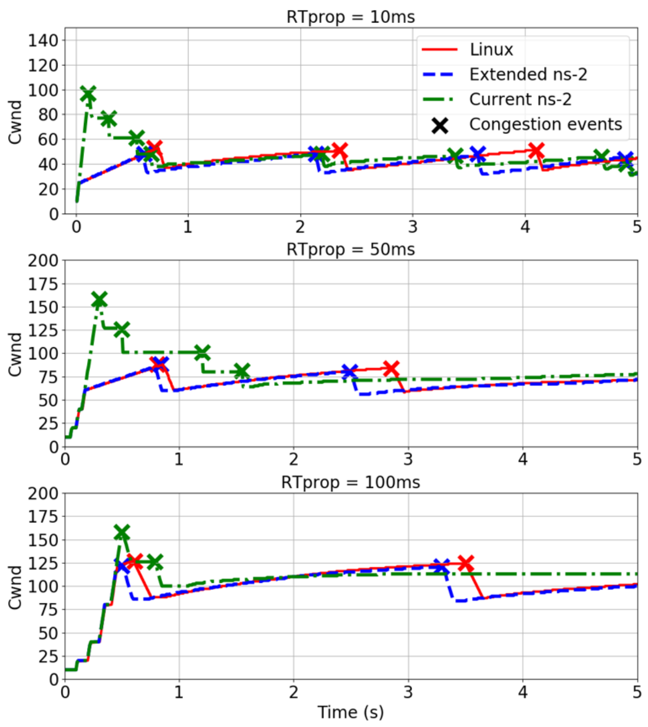

Figure 11: Comparison of cwnd trajectories with second-hop bottleneck of 10Mbps for round-trip propagation delays of 10ms, 50ms, and 100ms.

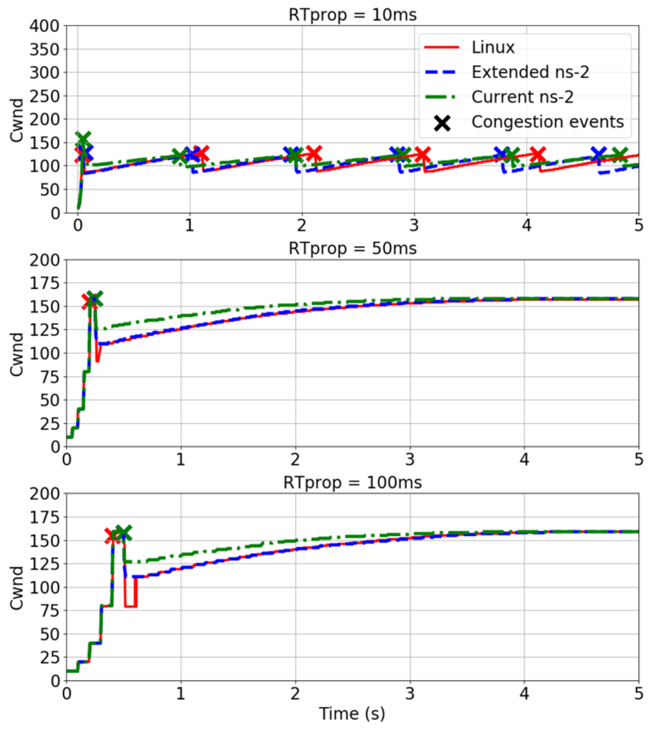

Figure 12: Comparison of cwnd trajectories with second-hop bottleneck of 100Mbps for round-trip propagation delays of 10ms, 50ms, and 100ms.

6 SUMMARY AND FUTURE WORK

This work uncovered the intra-host-back-pressure issue in current ns-2 implementation and developed extensions to substantially improve its fidelity in simulating recent Linux TCP implementation. In addition to the extensions investigated in this work, there are a number of TCP-related mechanisms which are absent from ns-2, most notably TCP Control Block [18], TCP Fast Open [25], TCP receiver window, etc. Although these mechanisms are outside the scope of TCP congestion control (the focus of many TCP studies) they could impact TCP behavior under certain network conditions. Therefore additional work is warranted to investigate their performance impact and implementation into ns-2.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their suggestions and comments in improving this paper to its final form.

REFERENCES

[1] Network Simulator-2 ns-2. [Online] http:www.isi.edu/nsnam/ns/

[2] I. Rhee and L. Xu. Cubic: A New TCP-Friendly High-Speed TCP Variant. *ACM SIGOPS Operating System, New York, USA*, Jul.2008, pp. 64-74.

[3] K. Tan, J. Song, Q. Zhang, and M. Sridharan. A Compound TCP Approach for High-Speed and Long-Distance Networks. *in Proc. IEEE INFOCOM' 2006*, Barcelona, Spain, Apr. 2006, pp 1-12.

[4] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data Center TCP (DCTCP). *in Proc. SIGCOMM'2010*, New Delhi, India, Aug. 2010, pp. 63-74.

[5] K. Winstein and H. Balakrishnan. TCP ex Machina: Computer-Generated Congestion Control. *in Proc. SIGCOMM'2013*, Hong Kong, China, Aug. 2013, pp 123-134.

[6] L. A. Grieco and S. Mascolo. Performance Evaluation and Comparison of Westwood+, New Reno, and Vegas TCP Congestion Control. *in Proc. SIGCOMM'2004*, Portland, Oregon, USA. Aug. 2004, pp. 25-38.

[7] D. X. Wei, P. Cao. ns-2 TCP-Linux: An ns-2 TCP Implementation with Congestion Control Algorithm from Linux. *in Proc. The Workshop on Ns-2: The IP Network Simulator (WNS2'2006)*, Pisa, Italy. Oct. 2006.

[8] S. Jansen and A. McGregor. Simulation with Real World Network Stacks. *In Proc. 37th Conference on Winter Simulation (WSC'05)*, Orlando, Florida, USA. Dec. 2005, pp. 2454-2463.

[9] Extended ns-2. [Online]. https://github.com/mclab-cuhk/Extended_ns2

[10] Network Simulator-3 ns-3. [Online]. https://www.nsnam.org/

[11] D. Siemon. Queueing in the Linux Network Stack. *Linux Journal*, Houston, TX, USA, Sep. 2013.

[12] S. Ha and I. Rhee. Taming the Elephants: New TCP Slow Start. *Computer networks*, vol. 55, issue. 9. pp. 2092-2110. Jun. 2011.

[13] R. Braden. Requirements for Internet Hosts—Communication Layers. *RFC 1122*, 1989.

[14] M. Handley, J. Padhye, S. Floyd. TCP Congestion Window Validation. *RFC 2861*, 2000.

[15] Better Follow Cubic Curve After Idle Period (Sep 2015). [Online] https://github.com/torvalds/linux/commit/30927520dbae297182990bb21d8762bcc35ce1d

[16] TCP Small Queues. [Online] https://lwn.net/Articles/507065/

[17] S. Floyd, M. Mathis, J. Mahdavi, and A. Romanow. TCP Selective Acknowledgement Option. *RFC 2018*, 1996.

[18] TOUCH, J. TCP Control Block Interdependence. *RFC 2140*, 1997.

[19] J. Chu, N. Dukkipati, Y. Cheng, and M. Mathis. Increasing TCP's Initial Window. *RFC 6928*, 2013.

[20] Ethtool. [Online] https://linux.die.net/man/8/ethtool

[21] Netem. [Online] http://man7.org/linux/man-pages/man8/tc-netem.8.html

[22] Iperf. [Online] https://iperf.fr/

[23] Tcpprobe.[Online] https://wiki.linuxfoundation.org/networking/tcpprobe

[24] Tcpdump. [Online] https://linux.die.net/man/8/tcpdump

[25] J, Chu. TCP Fast Open. *RFC 7413*, 2014.