Data Center TCP in ns-3: Implementation, Validation and Evaluation

Vivek Jain*
University of California, Riverside
Riverside, California, USA
vjain014@ucr.edu

Thomas R. Henderson
University of Washington
Seattle, Washington, USA
tomhend@uw.edu

Shravya K. S.
National Institute of Technology Karnataka, Surathkal
Mangalore, Karnataka, India
shravya.ks0@gmail.com

Mohit P. Tahiliani
National Institute of Technology Karnataka, Surathkal
Mangalore, Karnataka, India
tahiliani@nitk.edu.in

ABSTRACT

Data Center TCP (DCTCP) is a standard congestion control scheme used to provide high burst tolerance, low latency and high throughput in Data Center Networks (DCNs). It uses in-network feedback obtained through Explicit Congestion Notification (ECN) to scale the congestion window (*cwnd*). Recently, there have been significant efforts to extend DCTCP to provide low latency transport (e.g., TCP Prague) in a Wide Area Network, but with congestion control and in-network queuing mechanisms that are backward compatible with legacy implementations. This paper presents the implementation and validation of a new ns-3 model for DCTCP which can be used to implement its extensions, such as TCP Prague. The proposed model is verified and validated by comparing it to the Linux model of DCTCP by using ns-3 Direct Code Execution (DCE), a framework to run ns-3 simulations using Linux network stack. The results obtained from this comparison show that the ns-3 model and Linux model of DCTCP exhibit similar characteristics.

CCS CONCEPTS

• **Networks** → **Network experimentation**; *Network simulations*; Network performance analysis.

KEYWORDS

ns-3, Explicit Congestion Notification, Data Center TCP

ACM Reference Format:
Vivek Jain, Thomas R. Henderson, Shravya K. S., and Mohit P. Tahiliani. 2020. Data Center TCP in ns-3: Implementation, Validation and Evaluation. In *2020 Workshop on ns-3 (WNS3 2020), June 17–18, 2020, Gaithersburg, MD, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3389400.3389405

1 INTRODUCTION

Data Center traffic often comprises of bursty flows (e.g., mining advertisements), short flows (e.g., search queries), and long flows (e.g., software updates), where bursty flows require high burst tolerance, short flows require low latencies and long flows require high throughputs. The need to satisfy these requirements is apparent due to the widespread adoption of cloud based applications. Data Center TCP (DCTCP) [1] is a congestion control algorithm implemented in major operating systems and used in Data Centers [2]. Different studies have shown that, although DCTCP provides the desired performance in Data Center Networks (DCNs), it cannot be used in a Wide Area Network (WAN) without modifications [9]. The main concerns are: (i) it cannot coexist fairly with widely deployed TCP congestion control algorithms in WAN, e.g., Reno [5] and CUBIC [7], and (ii) it uses a modified ECN mechanism which cannot be used in WAN because it is unreliable and is only suitable for privately controlled networks such as DCNs. As a consequence, an evolution of DCTCP called TCP Prague has been proposed for WAN in the Transport Area Working Group (TSVWG) of IETF. This has led to an active interest in the performance modeling of DCTCP and TCP Prague. However, to our knowledge, simulation models do not exist for these congestion control algorithms.

ns-3 [8] is a widely used open-source network simulator which provides Linux-like implementation of TCP congestion control algorithms [10]. These simulation models help to study the behavioral aspects of new congestion control algorithms before they are considered to be safe for deployment on the Internet. The evaluation of network protocols in Data Center environments requires the use of simulation models because using real testbeds can be impractical or expensive. This paper makes three contributions: (a) enables the support of ECN mechanism for TCP and queue disciplines (qdiscs) in ns-3, (b) provides a new ns-3 model for DCTCP (including modified ECN semantics), and (c) presents a thorough and systematic comparison of the ns-3 model with the Linux model of DCTCP by using Direct Code Execution (DCE), a framework to run ns-3 simulations using Linux network stack [12]. The work done in (a) and (b) has been merged into the mainline of ns-3, and (c) provides detailed insights about the characteristics of DCTCP, besides validating the correctness of the proposed model.

The paper is organized as follows: Section 2 presents the detailed working of ECN and DCTCP. Section 3 describes the implementation of ECN and DCTCP in ns-3. Section 4 discusses different approaches used to validate the proposed models in ns-3. The working of DCTCP is evaluated in Section 5, and Section 6 concludes the paper with directions for future work.

*This work was done when Vivek Jain was a Masters Student at National Institute of Technology Karnataka, Surathkal, Mangalore, India.

WNS3 2020, June 17–18, 2020, Gaithersburg, MD, USA

ACM ISBN 978-1-4503-7537-5/20/06…$15.00
https://doi.org/10.1145/3389400.3389405

2 BACKGROUND

2.1 Explicit Congestion Notification

ECN is a standard mechanism described in RFC 3168 for network devices to notify endpoints of congestion that may be developing in a bottleneck queue, without resorting to packet drops. It uses 2 bits in the IP header: ECN Capable Transport (ECT) and Congestion Experienced (CE) as shown in Figure 1, and 2 bits in the TCP header: Congestion Window Reduced (CWR) and ECN Echo (ECE) as shown in Figure 2. The combination of respective 2 bits in an IP header or TCP header is known as a codepoint [11]. Table 1 and Table 2 show the ECN codepoints in the IP and the TCP header, respectively.

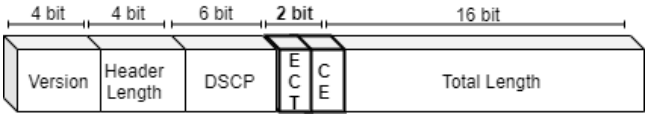


Figure 1: ECN Bits in the IP Header

Table 1: ECN Codepoints in the IP Header

Codepoint	ECT bit	CE bit
Non-ECT	0	0
ECT(1)	0	1
ECT(0)	1	0
CE	1	1

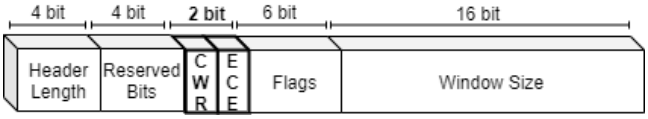


Figure 2: ECN Bits in the TCP Header

Table 2: ECN Codepoints in the TCP Header

Codepoint	CWR bit	ECE bit
Non-ECN-Setup	0	0
ECN-Echo	0	1
CWR	1	0
ECN-Setup	1	1

TCP sender and receiver negotiate the use of ECN during the three-way handshake as shown in Figure 3. If both are ECN capable, the sender sets either ECT(1) or ECT(0) codepoint in the IP header of every outgoing data packet, otherwise sets the Non-ECT codepoint. This serves as an indication to the router that both sender and receiver are ECN capable. Whenever router detects congestion build up (e.g., by using an Active Queue Management mechanism), it replaces the ECT(1) or ECT(0) codepoint by CE codepoint in the IP header. This process is called *marking the packet*. When the receiver receives a marked packet with CE codepoint, it infers congestion and hence, sends a series of outgoing acknowledgments (ACKs) with ECE codepoint in TCP header until the sender acknowledges with CWR codepoint. It is important to note that even if the router marks just one data packet with CE, the receiver sends a series of ACKs with ECE until it receives confirmation from the sender. This is to ensure the reliability of congestion notification; because even if the first ACK carrying ECE is lost, other ACKs with ECE would notify the sender about congestion. Figure 4 shows in brief, the steps involved in the working of ECN.

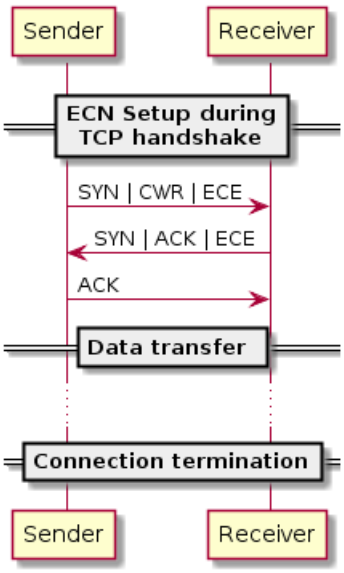


Figure 3: ECN Negotiation

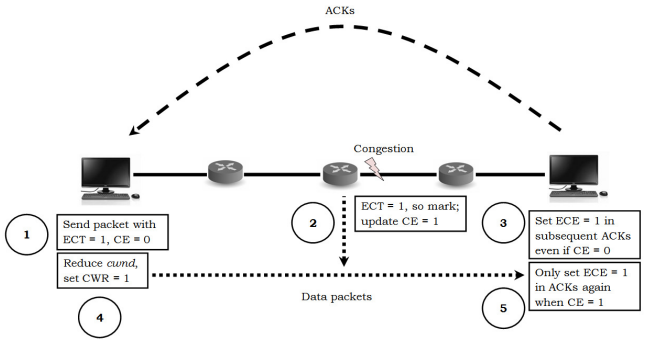


Figure 4: Working of ECN

2.2 Data Center TCP

DCTCP scales the congestion window (*cwnd*) depending on *the amount of congestion* in the network, which is estimated by using a modified ECN mechanism. Thus, there are three main components in the DCTCP algorithm: (i) intermediate network devices must detect congestion and set the CE codepoint in the IP header of the packet, (ii) DCTCP receiver must set ECE codepoint in ACK to

inform DCTCP sender about congestion and (iii) DCTCP sender must estimate *the amount of congestion* using this information and accordingly scale the *cwnd*. The details of every component are discussed below. However, if DCTCP detects a packet loss, it scales its *cwnd* in the same way as a typical TCP.

2.2.1 Setting CE Codepoint at Intermediate Devices. DCTCP requires the intermediate devices to use *current queue occupancy* as a measure to set CE codepoint in the incoming packets. On arrival of every packet, the *queue occupancy* is measured, and the packet is marked with CE codepoint if the queue is occupied more than a predetermined congestion threshold (K). It is recommended that:

$$K > \frac{(RTT \times C)}{7} \tag{1}$$

where, RTT is the Round Trip Time and C is the link rate in packets.

2.2.2 Modified ECN Semantics for the Receiver. On receiving a packet with CE codepoint, DCTCP receiver sends an ACK with ECE to inform the DCTCP sender about congestion. DCTCP receiver differs from a typical TCP receiver in two aspects:

(a) For reliability of ECN, a typical TCP receiver sends a series of ACKs with ECE even if only one packet is CE marked by the intermediate device. DCTCP receiver relaxes this reliability aspect of ECN and sets a ECE codepoint in ACK only when it receives the CE codepoint. This helps the sender to count the exact number of packets that were marked with CE at the intermediate device and estimate the *amount of congestion* in the network. It is believed that since DCTCP is designed to operate in privately controlled and well managed data centers, ACK losses are not common. Hence, the modified ECN semantics suffices. However, this aspect prevents DCTCP from being deployed in WAN and other lossy environments because loss of ACKs with ECE codepoint leads to inaccurate estimate of amount of congestion by the DCTCP sender.

(b) Delayed ACKs are an important optimization for data centers and are deployed in all major operating systems. When delayed ACKs are used, TCP receiver sends one ACK for every k packets (typically, $k = 2$). However, it is crucial for the DCTCP receiver to send an ACK with ECE without delay when it receives a packet with CE codepoint. This issue is tackled by configuring the DCTCP receiver to send an ACK immediately when it receives packets with CE codepoint, otherwise send a delayed ACK.

2.2.3 Estimating Congestion at the Sender. DCTCP sender follows an Additive Increase and Multiplicative Decrease (AIMD) algorithm to scale the *cwnd*. *cwnd* is additively increased as shown in Eq. (2) and multiplicatively decreased as shown in Eq. (3).

$$cwnd = cwnd + \frac{1}{cwnd} \tag{2}$$

$$cwnd = cwnd \times (1 - \frac{\alpha}{2}) \tag{3}$$

where α ($0 \leqslant \alpha \leqslant 1$) is the fraction of packets marked in the previous *cwnd* and is calculated as shown in Eq.(4). F in (4) is the fraction of packets that are marked in the previous *cwnd* and g ($0 < g < 1$) is the exponential weighted moving average constant. Thus, when congestion is low (α is near 0), *cwnd* is reduced slightly and when congestion is high (α is near 1), *cwnd* is reduced by half, just like traditional TCP. This aspect of DCTCP is another reason why deploying it in a WAN environment is challenging because it would starve other coexisting TCP flows that employ a fixed and larger multiplicative decrease.

$$\alpha = (1 - g) \times \alpha + g \times F \tag{4}$$

The *cwnd* is increased on arrival of every ACK, but it is decreased only once for *cwnd* i.e., when the first ACK with ECE is received by DCTCP sender for that *cwnd*.

2.2.4 Marking Control Packets. RFC 3168 prohibits the use of ECN for TCP control packets, such as SYN, SYN+ACK and RST i.e., these control packets cannot be marked with a CE codepoint by the intermediate devices. But RFC 8257 recommends using ECN for these control packets so that they are not dropped by the intermediate devices. DCTCP receiver, however, does not respond to CE codepoint on these control packets.

3 PROPOSED MODEL

3.1 Implementation of ECN

To enable support of ECN in ns-3, both TCP endpoints and intermediate devices must support the functionality of ECN. Hence, the implementation was done in two phases. In the first phase, the TCP endpoints were made compatible with the standard ECN algorithm. In the second phase, ECN functionality was added to the traffic control layer. This allows intermediate devices running queue discipline algorithms like Random Early Detection (RED) [6] to mark ECN bits in the IP Header.

Phase 1. ECN support has been added to the internet module of ns-3. At the time of implementation, it was noticed that ECN specific bits were already supported in `TcpHeader` and `Ipv4Header` in ns-3. `Ipv6Header` was extended to support ECT and CE bits. These bits were added in the last 2 bits of the Traffic class that occupies the 10th and 11th bit in the IPv6 header.

The ECN negotiation process shown in Figure 4 is implemented in the `TcpSocketBase` class. This requires setting `ECE` and `CWR` bits in the TCP Header. However, the `SetFlag()` method allowed setting only six flag bits of the standard TCP Header. Hence, `SetFlag()` was modified to support the setting of the two most significant bits that represent ECE and CWR in the TCP header.

The transport layer dictates the IP layer to set the ECT codepoints by using `SocketIpTosTag` and `SocketIpv6TclassTag` packet Tags for IPv4 and IPv6 headers, respectively. The ECN specific events described in RFC 3168, such as the reception of CE packets, are represented by ECN states defined in the `TcpSocketState` class. Table 3 provides a list of ECN states and their description. Figure 5 and Figure 6 depict the state transitions for the TCP sender and TCP receiver, respectively, and the details are as follows:

(1) Initially both sender and receiver are in NO_ECN.
(2) If ECN negotiation succeeds, their states change to ECN_IDLE.
(3) On receiving CE, receiver state changes to ECN_CE_RCVD.
(4) On sending ECE, receiver state changes to ECN_ECE_SENT.
(5) On receiving ECE, sender state changes to ECN_ECE_RCVD.
(6) When the sender sends CWR, its state changes to ECN_IDLE.
(7) On receiving CWR, receiver state changes to ECN_IDLE.

Table 3: List of ECN States in ns-3

ECN states	Description
NO_ECN	ECN is not negotiated, ECN disabled traffic
ECN_IDLE	ECN is enabled, currently there is no action
ECN_CE_RCVD	Receiver received a packet with CE codepoint
ECN_ECE_SENT	The receiver keeps sending an ACK with ECE until it receives CWR
ECN_ECE_RCVD	On receiving ECE, sender sends CWR

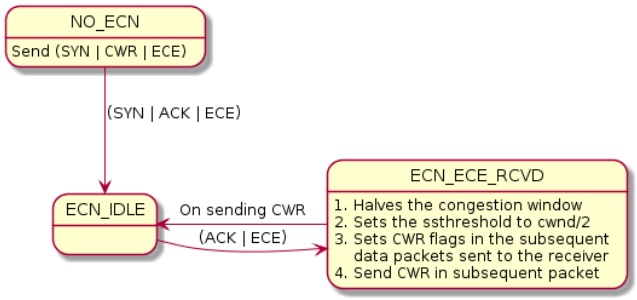


Figure 5: ECN State Machine for TCP Sender

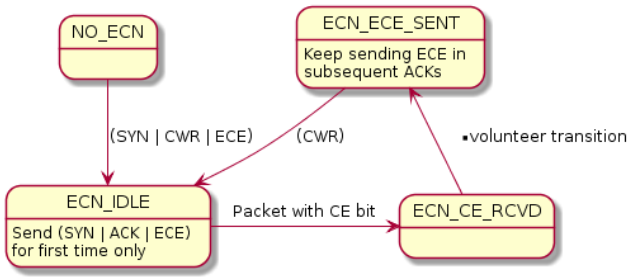


Figure 6: ECN State Machine for TCP Receiver

A Linux-like attribute to enable/disable ECN at the transport layer has been added in ns-3. By default, it is `Off` in the ns-3 TCP socket. It can be enabled by setting `ns3::TcpSocketBase::UseEcn`.

Phase 2. ECN functionality has been added in traffic control module of ns-3 so that intermediate devices can mark the packets. Since every packet is encapsulated as `QueueDiscItem`, we added a pure virtual method `Mark()` in `QueueDiscItem` and implemented it in `IPv4QueueDiscItem` and `IPv6QueueDiscItem`. To demonstrate the ECN usage, `DoEnqueue()` method of `RedQueueDisc` class is extended to support marking. Similar to `TcpSocketBase`, an attribute called `UseEcn` is added in the `RedQueueDisc` class to enable/disable ECN. This model of ECN is available since ns-3.29, and follows the recommendations of RFC 3168:

(1) Pure ACK packets should not have ECT bit set.
(2) Retransmitted packets should not have ECT bit set in order to prevent Denial of Service (DoS) attack.
(3) Sender should reduce the *cwnd* only once in each window.
(4) Sender and Receiver should ignore the ECE bit and CE bit, respectively in the packets that arrive out of the window.

3.2 Implementation of DCTCP

The implementation of DCTCP in ns-3 relies upon the methods of `TcpCongestionOps`, which is a Linux-like interface used for implementing congestion control algorithms in ns-3. This is similar to how other congestion control algorithms are currently implemented in ns-3. `TcpDctcp` class is inherited from the `TcpNewReno` because it retains some functionalities of Newreno (e.g., slow start, additive increase, fast retransmit and fast recovery). We provide the implementation of `pure virtual` methods and override the virtual methods to develop the DCTCP model. While the implementation process is similar to other TCP algorithms, DCTCP has the following exceptions: it has a strict dependence on the ECN mechanism, uses a modified ECN mechanism, permits the use of ECN on SYN, SYN+ACK and RST packets, uses delayed ACKs differently and scales *cwnd* based on *the amount of congestion.*

3.2.1 Prerequisite Functionality. ECN is disabled in ns-3 by default. But since it is mandatory for DCTCP, a new virtual method called `Init()` has been implemented in `TcpCongestionOps` class to auto-enable ECN for DCTCP. This is important because a ns-3 user may not be aware of the tight coupling between ECN and DCTCP. `Init()` is similar to the `init` function in Linux, which initializes the state information of congestion control algorithms. This method is not specific to the implementation of DCTCP and can be used by other congestion control algorithms in ns-3, e.g., it can be used to auto-enable pacing when TCP BBR [3] is used. Yet another generic (and `virtual`) method that has been added in `TcpCongestionOps` class is `ReduceCwnd()`. This method is used to reduce the *cwnd* when an ACK arrives with ECE codepoint. DCTCP overrides this method to implement its *cwnd* reduction policy shown in Eq. (3). Similarly, this method can be used for TCP BBRv2 [4] in future because it follows a DCTCP-style *cwnd* reduction policy.

3.2.2 Sender Side Functionality. DCTCP sender uses Eq. (2)-(4) to scale *cwnd*. The default value for g is 0.0625 and α is initialised to 1 as recommended in RFC 8257. These parameters are user configurable and added as attributes in ns-3. Another attribute is `UseEct0` which allows the user to select one among the two ECT codepoints to be used in IP header. DCTCP sender sets ECT codepoints on SYN, SYN+ACK, RST besides data packets. If ECN is not used or when a packet loss is detected, DCTCP sender behaves same as Newreno.

3.2.3 Receiver Side Functionality. As discussed in Section 2.2.2, DCTCP receiver uses modified semantics of ECN. Figure 7 is a modification of Figure 6 showing the differences between original ECN and modified ECN mechanism for DCTCP receiver. This modification ensures that an ACK with ECE is sent only when a CE codepoint is received by the DCTCP receiver. Like Linux, `CwndEvent()` with `CA_EVENT_ECN_NO_CE` and `CA_EVENT_ECN_IS_CE` events is used to track the CE codepoint. Depending on these events, DCTCP receiver sends ECE. Similarly, the `CwndEvent()` method along with `CA_EVENT_DELAYED_ACK` and `CA_EVENT_NON_DELAYED_ACK` events is used for delayed ACKs. These events allow the DCTCP receiver to exhibit the functionality mentioned in Section 3.2 of RFC 8257.

To send ACKs, `TcpDctcp` class needs to call `SendEmptyPacket()` method of the `TcpSocketBase` class. This is achieved by adding a callback for the required method in `TcpSocketState` and it is used for sending acknowledgments from the `TcpDctcp` class.

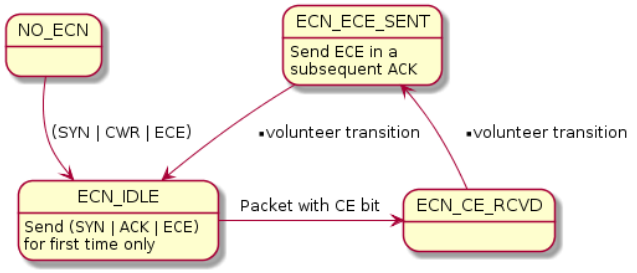


Figure 7: ECN State Machine for DCTCP Receiver

3.2.4 Switch/Router Side Functionality. DCTCP depends on a simple queue management algorithm in switches/routers to mark packets if the current queue occupancy exceeds a predetermined congestion threshold (See Eq. (1)). The current implementation of DCTCP in ns-3 uses RED with a simple configuration to achieve the behavior of desired queue management algorithm: minimum and maximum threshold of RED are set to the same value, and the queue weight used during the calculation of average queue length in RED is set to 1 to take into consideration the instantaneous queue length only.

This model has been tested extensively against a version of DCTCP in the Linux kernel (v4.4) using ns-3 DCE. The DCTCP model presented in this paper is merged in the development branch of ns-3, and will be available in the upcoming release (ns-3.31).

4 VALIDATION

4.1 Unit Test Cases

We validate the correctness of ECN and DCTCP models proposed for ns-3 by developing a separate test suite for each.

4.1.1 Test Suite for ECN. This test suite validates the standalone features of ECN in ns-3: Test 1 verifies the final ECN states after TCP handshake, when both sender and receiver do not support the ECN. The expected state is `NO_ECN`. Test 2 verifies the final ECN states after TCP handshake, when the sender is ECN capable but the receiver is not. The expected state is `NO_ECN`. Test 3 verifies the final ECN states after TCP handshake, when the receiver is ECN capable but the sender is not. The expected state is `NO_ECN`. Test 4 verifies the final ECN states after TCP handshake, when both sender and receiver support ECN. The expected state is `ECN_IDLE`. Test 5 verifies the receiver's behavior of noticing the CE marked packet and sending the ECE packets until it receives the CWR packet. Test 6 verifies the sender's behavior of reducing the *cwnd* on receiving ECE in ACK and sending CWR in subsequent data packet.

4.1.2 Test Suite for DCTCP. This test suite is developed to validate the working of DCTCP model in ns-3: Test 1 verifies the *cwnd* decrease policy used by DCTCP sender on receiving ECE. Test 2 verifies the DCTCP receiver's functionality of sending ECE packets only when CE is received. Test 3 verifies the compliance with Section 3.6 of RFC 8257 which states that DCTCP sender must set ECT codepoints for SYN, SYN + ACK and RST packets. Test 4 verifies whether ns-3 model of DCTCP behaves like NewReno during the Slow Start phase.

4.2 Comparison with Linux DCTCP using DCE

DCE is a framework over ns-3 that provides a feature to run simulations using real Linux network stack. Besides, it provides a feature to use real network applications (such as iperf) inside ns-3 environment without changing their code. Our validation uses the latest Linux kernel supported by DCE (v4.4). We updated the Linux kernel to scrape the *cwnd* and patched the "low alpha" bug that was discovered by the community in Linux DCTCP. The Linux DCTCP implementation uses integer arithmetic and a low resolution for α, with the result that when α is close to zero, it rounds to zero (and thereby disables a *cwnd* response) until the marking level rises above roughly 10%. Furthermore, we implemented Linux-like pacing because ns-3 supports static pacing only. This is useful to achieve low latency behaviour in ns-3 DCTCP model.

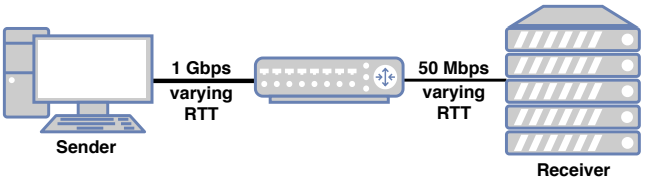


Figure 8: Topology used for DCTCP Validation

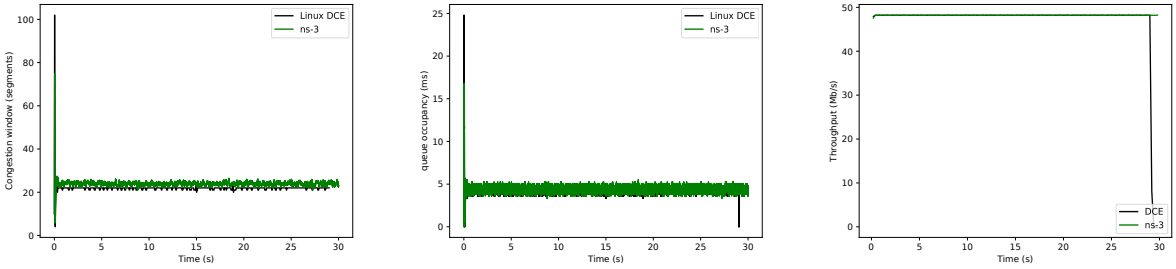
The validation scenario comprises one TCP flow as shown in Figure 8. Other simulation parameters are listed in Table 4. The base RTT is varied and the metrics considered for validation are: sender's *cwnd*, throughput and queue depth at bottleneck router.

Table 4: Simulation Parameters

Parameter	Value
Bottleneck Bandwidth	50 Mbps
SegmentSize	1448 bytes
SndBufSize/RcvBufSize	8192000 bytes
InitialCwnd	10 segments
EnablePacing	true
PaceInitialWindow	true
UseEcn	true
ARED	true
MinTh and MaxTh	17 and 34 packets
QW	1

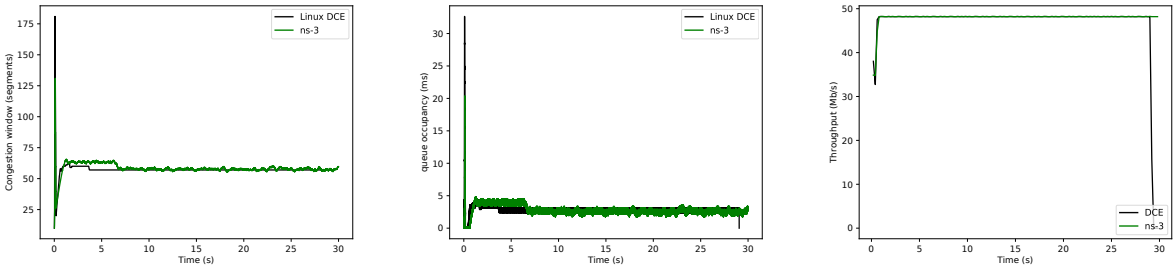
Figures 9-11 show the results for three values of base RTT: 3 μs, 10 ms and 50 ms, respectively. These results confirm that the DCTCP model in ns-3 has the same characteristics as that of Linux. However, some differences are observed during the Slow Start phase, which are depicted in Figures 12- 14 (zoomed plots). The reasons for these variations are as follows: (1) ns-3 uses byte-level granularity to maintain TCP *cwnd* whereas Linux uses segment-level granularity. Linux does not use float/double based inflation of the *cwnd*. Recently, a new class called `LinuxReno` was developed in ns-3 during Google Summer of Code 2019[1] which uses segment-level granularity to maintain *cwnd* in ns-3. We aim to test the working

[1] https://ns-3-tcp-validation.github.io/



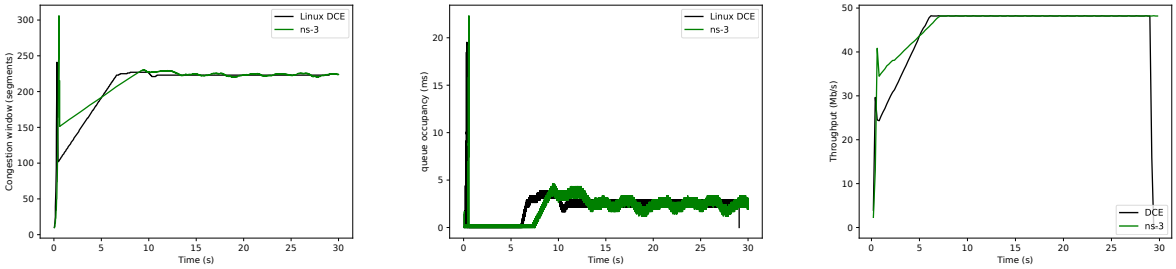
(a) Congestion Window **(b) Queue Length** **(c) Throughput**

Figure 9: 3μs RTT



(a) Congestion Window **(b) Queue Length** **(c) Throughput**

Figure 10: 10ms RTT



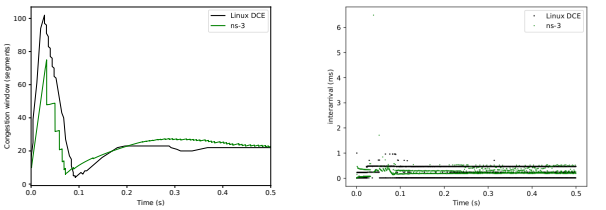
(a) Congestion Window **(b) Queue Length** **(c) Throughput**

Figure 11: 50ms RTT

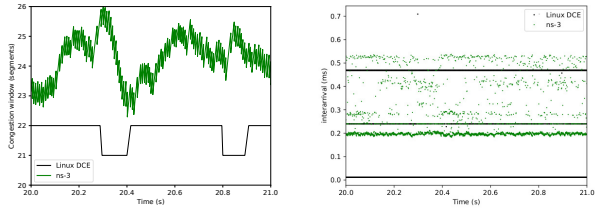
of proposed DCTCP model by using segment-level granularity to maintain *cwnd* during the slow start phase after TCP LinuxReno model is merged into the mainline of ns-3. (2) Pacing in ns-3 is implemented as follows: For values of *cwnd* less than slow start threshold/2, the window is paced out across half of the measured RTT, and for other values, the window is paced out across 80% of the measured RTT; this policy is based on current Linux internal pacing defaults. Moreover, ns-3 avoids back-to-back transmissions (after initial window), but Linux usually performs back-to-back and only when it has to send more than 2 segments, it paces the third. (3) DCTCP in Linux reuses the CWR state (from fast recovery) and uses Proportional Rate Reduction (PRR), whereas ns-3 TCP does not support `CA_CWR` state yet. (4) For larger values of *cwnd* and longer RTTs, the Linux code can suffer from *cwnd* latching for some values of steady state marking probability (Figure 13(c) and Figure 14(c)); the reduction only affects the integer part of the *cwnd* value and not the MSS counter, and rounds to the same value every RTT. This is probably not observed in the data center because RTT is small.

4.3 Fallback to Newreno

This experiment verifies that DCTCP safely falls back to Newreno when the intermediate devices drop packets instead of marking.

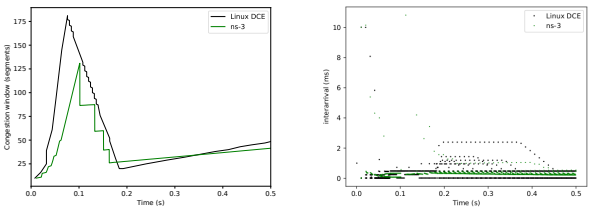


(a) *cwnd* (Slow Start) (b) Interarrival (Slow Start)

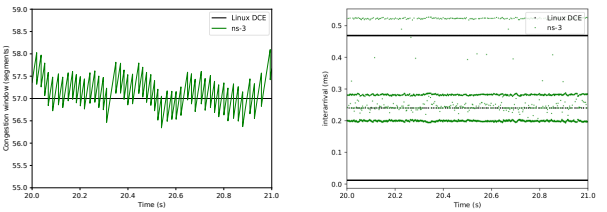


(c) *cwnd* (Steady State) (d) Interarrival (Steady State)

Figure 12: 3μs RTT (zoomed)



(a) *cwnd* (Slow Start) (b) Interarrival (Slow Start)



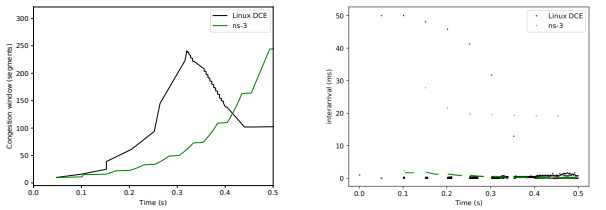
(c) *cwnd* (Steady State) (d) Interarrival (Steady State)

Figure 13: 10ms RTT (zoomed)

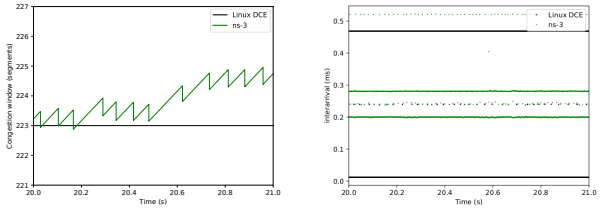
The experiment uses the same topology shown in Figure 8. The marking capability at the intermediate device is disabled. Figures 15 and 16 confirm that the proposed DCTCP model safely falls back to Newreno when intermediate devices do not mark the packets.

5 EVALUATION

We use the scenario described in Figure 17 of [1] to evaluate the characteristics of DCTCP. As shown in Figure 17, we use a topology consisting of forty DCTCP flows, and three switches to examine throughput, fairness, and queue delay properties of the network.



(a) *cwnd* (Slow Start) (b) Interarrival (Slow Start)



(c) *cwnd* (Steady State) (d) Interarrival (Steady State)

Figure 14: 50ms RTT (zoomed)

The second switch does not contribute to marking because it has a 10 Gbps input and output link. The RED queues are configured for ECN marking as suggested in [1]. Table 5 compares the throughput and fairness obtained by DCTCP authors [1] and the ns-3 model. Jain's Fairness Index is used to calculate the fairness. The results show that DCTCP achieves high link utilization, low queue delay (not shown due to space constraints) and fairness. However, it is observed that throughput distributions and queue delays are very sensitive to RED parameters, same as noted by DCTCP authors.

Table 5: Throughput and Fairness

	S1 flows	S2 flows	S3 flows	Fairness
As per [1]	46 Mbps	475 Mbps	54 Mbps	0.99
ns-3 model	23 Mbps	471 Mbps	74 Mbps	0.99

The results presented in this paper can be reproduced by following the instructions provided on our code repository[2].

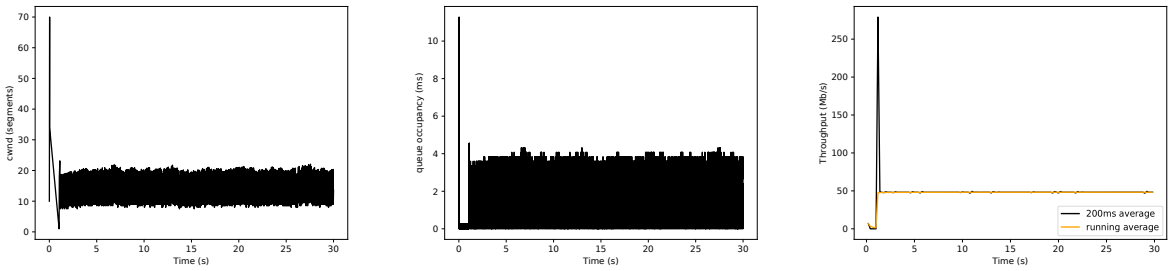
6 CONCLUSIONS AND FUTURE WORK

This paper presented the design and implementation of ECN and DCTCP in ns-3. Subsequently, an extensive validation of the proposed DCTCP is carried out by comparing it with its Linux implementation by using ns-3 Direct Code Execution (DCE). Further, the effectiveness of DCTCP has been evaluated in terms of controlling queue delay, utilizing the available bandwidth and achieving fairness with other competing flows. This model in ns-3 can be extended to implement TCP Prague in future.

ACKNOWLEDGMENTS

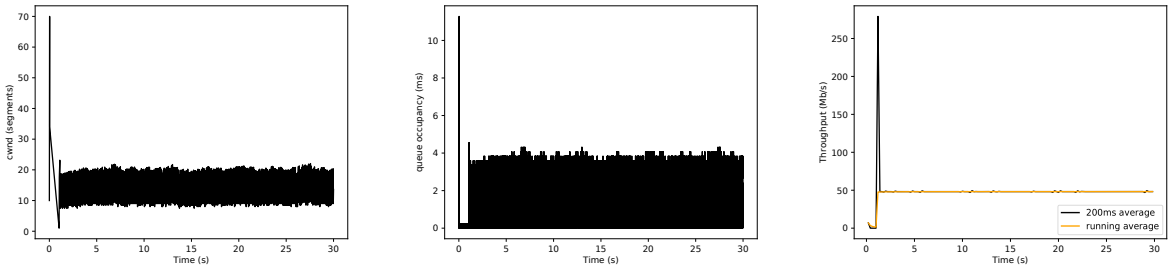
This paper is based on an original ns-3 ECN and DCTCP implementation by Shravya K. S., during ns-3 Summer of Code 2016 and

[2]https://github.com/Vivek-anand-jain/Reproduce-DCTCP-Results-in-ns-3



(a) Congestion Window (b) Queue Length (c) Throughput

Figure 15: Newreno without ECN



(a) Congestion Window (b) Queue Length (c) Throughput

Figure 16: DCTCP without ECN

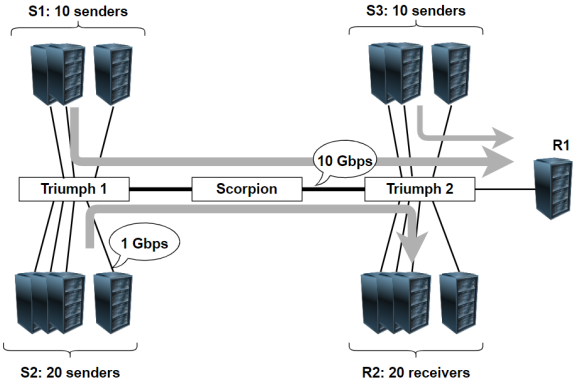


Figure 17: Multi-bottleneck Topology [1]

Google Summer of Code 2017, respectively. Vivek Jain wrote the supporting TCP ECN-handling code and was heavily involved in finalizing the DCTCP code that was added to the ns-3 mainline in early 2020. Shikha Bakshi and Apoorva Bhargava wrote the initial version of the large scale validation program, later finalized by Thomas Henderson, who also wrote the validation and plotting scripts. A bug in the Linux DCTCP code for low alpha values was fixed based on a patch by Bob Briscoe. Koen De Schepper provided insight on the Linux *cwnd* latching behavior, and Olivier Tilmans suggested the design for the dynamic pacing model.

REFERENCES

[1] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. 2011. Data Center TCP (DCTCP). *ACM SIGCOMM Computer Communication Review* 41, 4 (2011), 63–74.

[2] S. Bensley, L. Eggert, D. Thaler, P. Balasubramanian, and G. Judd. 2017. Data Center TCP (DCTCP): TCP Congestion Control for Data Centers. *RFC 8257, Internet Engineering Task Force* (2017).

[3] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson. 2017. BBR: Congestion-based Congestion Control. *Commun. ACM* 60, 2 (2017), 58–66.

[4] N. Cardwell, Y. Cheng, S. H. Yeganeh, I. Swett, V. Vasiliev, P. Jha, Y. Seung, M. Mathis, and V. Jacobson. 2019. BBRv2: A Model-Based Congestion Control. In *Presentation in ICCRG at IETF 104th meeting.*

[5] K. Fall and S. Floyd. 1996. Simulation-based Comparisons of Tahoe, Reno and SACK TCP. *ACM SIGCOMM Computer Communication Review* 26, 3 (1996), 5–21.

[6] S. Floyd and V. Jacobson. 1993. Random Early Detection Gateways for Congestion Avoidance. *IEEE/ACM Transactions on Networking* 1, 4 (1993), 397–413.

[7] S. Ha, I. Rhee, and L. Xu. 2008. CUBIC: A New TCP-Friendly High-Speed TCP Variant. *ACM SIGOPS Operating Systems Review* 42, 5 (2008), 64–74.

[8] T. R. Henderson, M. Lacage, G. F. Riley, C. Dowell, and J. Kopena. 2008. Network Simulations with the ns-3 Simulator. *ACM SIGCOMM Demonstration* 14, 14 (2008), 527.

[9] M. Kühlewind, D. P. Wagner, J. M. R. Espinosa, and B. Briscoe. 2014. Using Data Center TCP (DCTCP) in the Internet. In *2014 IEEE GLOBECOM Workshops (GC Wkshps).* IEEE, Austin, TX, USA, 583–588.

[10] V. Mittal, V. Jain, and M. P. Tahiliani. 2018. Proportional Rate Reduction for ns-3 TCP. In *Proceedings of the 10th Workshop on ns-3.* NITK Surathkal, Mangalore, India, 9–15.

[11] K. K. Ramakrishnan, S. Floyd, and D. Black. 2001. The Addition of Explicit Congestion Notification (ECN) to IP. *RFC 3168, Internet Engineering Task Force* (2001).

[12] H. Tazaki, F. Uarbani, E. Mancini, M. Lacage, D. Camara, T. Turletti, and W. Dabbous. 2013. Direct Code Execution: Revisiting Library OS Architecture for Reproducible Network Experiments. In *Proceedings of the Nineth ACM Conference on Emerging Networking Experiments and Technologies.* Santa Barbara, USA, 217–228.