Design and Implementation of TCP BBR in ns-3

Vivek Jain, Viyom Mittal, Mohit P. Tahiliani
Wireless Information Networking Group (WiNG)
Department of Computer Science and Engineering
NITK, Surathkal, Mangalore, India 575025
jain.vivek.anand@gmail.com, viyommittal@gmail.com, tahiliani@nitk.ac.in

ABSTRACT

Bottleneck Bandwidth and Round-trip propagation time (BBR) is a congestion based congestion control algorithm recently proposed by Google. Although it can be deployed with any transport protocol that supports data delivery acknowledgement, BBR is presently implemented alongside TCP (known as TCP BBR) in Linux kernel since 4.9 and is the default congestion control used in Google Cloud Platform. However, to the best of our knowledge, TCP BBR is not yet supported in popular network simulators such as ns-3. This limitation is a major hindrance in thoroughly studying the benefits of TCP BBR since carrying out large-scale and real-time experimental evaluations is a non-trivial task. In this paper, we discuss the design and implementation of a new model for TCP BBR in ns-3. We validate the proposed model by performing different sets of simulations to ensure that the model in ns-3 exhibits key characteristics of TCP BBR.

CCS CONCEPTS

• **Networks → Transport protocols**; **Network simulations**; • **Computing methodologies → Model development and analysis**;

KEYWORDS

ns-3, TCP, Congestion Control, BBR

ACM Reference Format:
Vivek Jain, Viyom Mittal, Mohit P. Tahiliani. 2018. Design and Implementation of TCP BBR in ns-3. In *Proceedings of the 2018 Workshop on ns-3 (WNS3 2018), June 2018, Surathkal, India.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3199902.3199911

1 INTRODUCTION

Congestion Control (CC) algorithms are essential components of TCP and play a vital role in restoring network stability after congestion epochs. The success of the Internet can be partly attributed to TCP because most of the Internet applications rely on TCP as the underlying transport protocol. However, the diverse characteristics and demands of the modern Internet applications have led to a series of optimizations being proposed to the congestion control mechanisms used by TCP. Majority of the CC algorithms that are presently used in the Internet (e.g., Reno [7], CUBIC [9]) assume packet loss as an indication of network congestion. This assumption is not reasonable because packet loss is not equivalent to congestion and can happen due to events other than congestion, e.g., routing failure or hardware bugs.

TCP BBR is a new congestion control algorithm developed by Google which takes a different approach to handle network congestion. Instead of assuming packet loss to be equivalent to congestion, TCP BBR builds an understanding of the network path in terms of maximum available bandwidth and minimum RTT to respond to actual congestion. Recent experimental studies are focusing on developing an understanding about the performance benefits offered by TCP BBR against widely deployed loss based CC algorithms. However, the implementation of TCP BBR is publicly available as a CC module for Linux only. To the best of our knowledge, TCP BBR is not supported in the native distributions of popular network simulators like ns-2 [8] and ns-3 [10]. This limitation poses challenges for researchers as they are left with few choices and have to mostly rely on real-time evaluations using Linux.

Recently, there have been significant efforts to align the implementation of TCP in ns-3 to that of Linux [5]. The native distribution of ns-3 contains several TCP congestion control algorithms, but unfortunately lacks support for TCP BBR. Furthermore, a few important features (e.g., pacing) that are prerequisites to implement TCP BBR are not supported in ns-3. In line with the efforts of the ns-3 development team and considering the interests of the research community, this paper makes the following two contributions: (i) we implement the prerequisite features that are necessary to implement TCP BBR in ns-3, and (ii) we implement a new model for TCP BBR in ns-3 which is based on its Internet draft [3]. The architecture of our proposed model follows the implementation of TCP BBR in Linux.

The outline of this paper is as follows: Section 2 gives a brief overview of BBR algorithm and its implementation in Linux. Additionally, it provides a brief background about the existing TCP models in ns-3. Section 3 highlights the prerequisite features that are mandatory to enable the implementation of BBR in ns-3. Section 4 discusses the design and implementation of TCP BBR model in ns-3. Section 5 presents the validation and functional verification of the results obtained from the proposed model with those results presented in the BBR paper. Finally, section 6 concludes the paper with directions for future work.

2 BACKGROUND

This section presents an overview of the TCP BBR algorithm and its implementation in the Linux stack. In addition, we discuss the current architecture of congestion control algorithms supported in ns-3 and highlight the list of prerequisite features that should be implemented in ns-3 before implementing TCP BBR.

WNS3 2018, June 2018, Surathkal, India

ACM ISBN 978-1-4503-6413-3/18/06…$15.00
https://doi.org/10.1145/3199902.3199911

2.1 Overview of TCP BBR

TCP BBR is a sender side congestion control algorithm which builds an explicit model of the network path and reacts to congestion accordingly. This model estimates the following two parameters: (i) available bottleneck bandwidth for a flow based on delivery rate estimation, i.e., the amount of data delivered to the receiver and (ii) the minimum round trip propagation delay based on the RTT measurements. Depending on these estimates that are obtained from the network model, TCP BBR tries to converge at an optimal operating point where the maximum available bandwidth of a path is utilized, and the RTT is minimum.

Unlike traditional CC algorithms that use only congestion window *cwnd*, BBR uses the following three distinct but interrelated parameters to control the sending behavior of a TCP connection:

- *pacing rate*: the rate at which a TCP sender sends data.
- *send quantum*: The maximum size of packet containing aggregated data that the transport sender is allowed to transmit over the network. This aggregation helps in reducing per-packet transmission overheads.
- *cwnd*: TCP sender's limit on the maximum amount of unacknowledged data in flight.

With the help of these control parameters and network model, TCP BBR aims to achieve two conditions, *Rate balance* i.e., packet arrival rate at bottleneck equals to bottleneck bandwidth and *Pipe full* i.e., total in-flight data is equal to the Bandwidth-Delay Product (BDP). *Rate balance* is achieved by pacing the packets close to the maximum bandwidth whereas *pipe full* is achieved by adapting pacing rate such that in-flight data stays near to the calculated BDP.

BBR operates in four different states wherein its control parameters (*pacing rate, quantum, cwnd*) are adapted to achieve high throughput and low latency. Figure 1 depicts the relationship and control flow among the different states of TCP BBR. The following is a brief description of every state:

- *Startup*: This is an initial state of TCP BBR algorithm. In this state, it ramps up the *cwnd* of the connection exponentially to find maximum available bandwidth on the path. This state is similar to slow start phase of the conventional TCP and might result in queue build-up at the intermediate routers.
- *Drain*: After observing maximum bandwidth, TCP BBR enters into the Drain state where it reduces its sending rate to evacuate any queue that was built up during Startup state.
- *ProbeBW*: TCP BBR spends most of the time in this state. It randomly updates the value of *pacing_gain* using a technique called *gain cycling with randomization* and checks for the higher value of bandwidth than the existing maximum value.
- *ProbeRTT*: In this state, TCP BBR intentionally reduces its *cwnd* to probe for minimum RTT.

2.2 BBR Implementation in Linux

Linux provides an interface handler called `struct tcp_congestion_ops` which provides the function pointers and is used to implement CC algorithms as pluggable modules. CC algorithms can override these function pointers and implement their custom behavior. The implementation of TCP BBR in Linux follows a similar approach and overrides the following function pointers that are provided by `struct tcp_congestion_ops`:

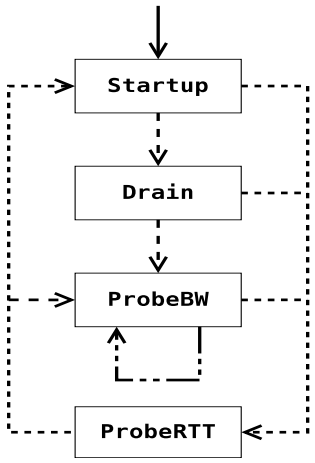


Figure 1: BBR state diagram [3]

- `void (*cong_control)(struct sock *sk, const struct rate_sample *rs)`: This function is called when packets are acknowledged and is used to update the *cwnd* and *pacing rate*. It performs the core functionality of BBR algorithm. BBR uses this function to obtain the latest information about the delivery rate and updates its network model (maximum available bandwidth and minimum RTT) and control parameters (*pacing rate, send quantum, cwnd*). This method is also responsible for BBR state transition based on the updated network information.
- `void (*set_state)(struct sock *sk, u8 new_state)`: This function is called when CA state changes (e.g., from `CA_OPEN` to `CA_DISORDER` on receipt of a DupAck). BBR implementation uses this method on CA_LOSS state (i.e., when a retransmission timeout happens) to save the current value of *cwnd*.
- `void (*cwnd_event)(struct sock *sk, enum tcp_ca_event ev)`: This function is called when a *cwnd* event occurs (e.g., when delayed ack is sent). BBR uses this method to update the pacing rate at the time of first packet transmission when no data is in flight.
- `u32 (*ssthresh)(struct sock *sk)` This function updates the value of *ssthresh* when required and returns its value. In case of BBR, it saves the current *cwnd* and returns the maximum integer value. BBR does not use the value of *ssthresh* and hence, never reduces it.
- `void (*init)( )` This function is used for initializing private variables of CC algorithms and its functionality is same for BBR.

Two important features are required for implementing BBR: pacing and delivery rate estimation. The initial implementation of TCP BBR in Linux was based on the pacing provided by the Fair Queue *(FQ)* scheduler. However, this approach restricted the usage of BBR with other scheduling algorithms. Therefore, a new approach for pacing has been implemented at the transport layer in Linux. This provides flexibility to the user to choose between the two approaches of pacing. As a default setting, BBR falls back to

transport layer pacing if the underlying scheduling algorithm being used is not *FQ*. Delivery rate estimation algorithm has been implemented for Linux TCP as per the guidelines provided in [3]. The delivery rate estimates are provided to BBR via the `cong_control` function.

2.3 TCP Model in ns-3

The *internet* module of ns-3 contains the implementation of TCP related functionalities. ns-3 follows a Linux-like approach to implement the CC algorithms [5]. The implementation is divided mainly into the following two classes: `TcpSocketBase` and `TcpCongestionOps`. This modular architecture simplifies the task of implementing new CC algorithms in ns-3.

`TcpSocketBase` is the main class for TCP which manages the TCP state machine and supports different features such as Limited Transmit [1], Loss Recovery [13], Window Scaling [11], Selective ACKnowledgements [12], etc. `TcpCongestionOps` is an abstract class which is similar to Linux's `tcp_congestion_ops` and can be inherited by different CC algorithms to implement their custom functionality. The following methods of `TcpCongestionOps` can be overridden by respective CC algorithms:

- `PktsAcked`: This method is called on receipt of an acknowledgment and it is similar to the `pkts_acked` method of Linux.
- `IncreaseWindow`: This is used to update *cwnd* and it is similar to the `cong_avoid` method of Linux.
- `CongestionStateSet`: This method is called before changing the congestion state and it is similar to the `set_state` method of Linux.
- `CwndEvent`: This method is invoked when *cwnd* event occurs and it is similar to the `cwnd_event` method of Linux.
- `GetSsthresh`: This method is used to fetch the value of *ssthresh* after a loss event and it is similar to the `ssthresh` method of Linux.

Different types of TCP CC algorithms such as loss based (e.g., NewReno), delay based (e.g., Vegas), Hybrid (e.g., Illinois), less than best effort (e.g., LEDBAT) are currently supported in the native distribution of ns-3. However, the functionality of BBR is significantly different than the conventional CC algorithms mentioned above. We found that the `TcpCongestionOps` class of ns-3 does not contain some methods that are required for implementing TCP BBR e.g., method equivalent to `cong_control` of Linux is not supported. Moreover, we observed that pacing, delivery rate estimation and windowed min-max filter techniques are missing in ns-3. Hence, in this work, we first extend the `TcpCongestionOps` class of ns-3 by implementing the necessary methods, followed by the implementation of pacing, delivery rate estimation and windowed min-max filter in ns-3. Subsequently, we implement a new model BRR in ns-3 and validate its functionality.

3 PREREQUISITE FEATURES FOR TCP BBR IN NS-3

This section describes our implementation of prerequisite features for TCP BBR in ns-3. The source code for these features has been submitted for review to the ns-3 development team, and some of the code has been merged in the mainline of ns-3 recently.

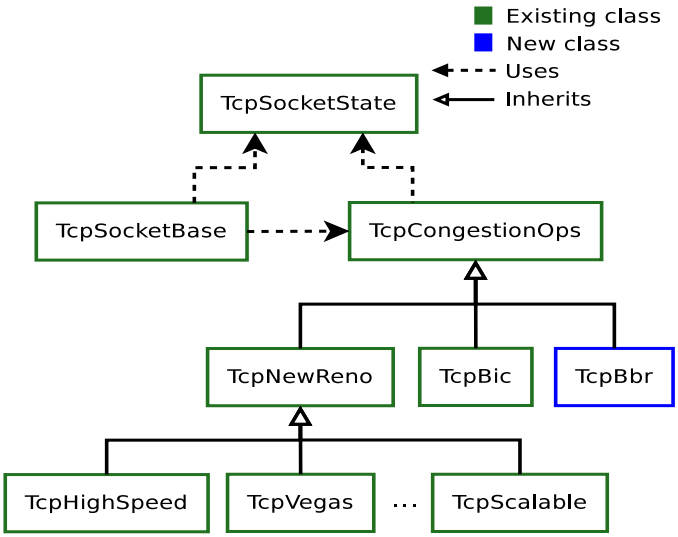
(1) Extending `TcpCongestionOps`: `TcpCongestionOps` class in ns-3 lacks a few methods which are crucial for implementing TCP BBR. Moreover, these methods have several other use cases besides being useful for implementing TCP BBR. We extended the `TcpCongestionOps` class by implementing the following methods. The source code for this work is under review and can be found here[1].
   (a) `bool HasCongControl ( ) const`: This method returns true if congestion control is using `CongControl ( )` instead of `IncreaseWindow ( )`.
   (b) `void CongControl (Ptr<tcpSocketState> tcb, const struct RateSample * rs)`: This method is similar to the `cong_control` method of Linux and is used to update the *cwnd* and *pacing rate* based on the estimation of delivery rate.
(2) Pacing: BBR uses *pacing* to control the rate of sending data into the network. The technique of pacing can be implemented either at the traffic control layer or at transport layer depending as done in Linux. However, ns-3 lacks the implementation of pacing in either of the layers. Hence, we implemented pacing in ns-3 at the transport layer by taking [6] as the reference. This work has been merged in `ns-3-dev` (changeset 13211:ebd04839f914). An attribute called "Pacing" has been added to the `TcpSocketState` class in ns-3 so that it can be enabled or disabled. Additionally, it is also possible to set the maximum pacing rate using our implementation.
(3) Delivery rate estimation [4]: BBR uses this algorithm to measure the approximate value of the delivery rate of in-flight data. This involves changes at sender side by maintaining per-packet information. Taking [4] as reference, we implemented rate estimation algorithm in `TcpTxBuffer` class of ns-3. The source code for this work is under review and can be found here[1].
(4) Windowed min-max filter: Another challenge faced during the implementation of BBR is that ns-3 does not have support for the windowed-min-max filter. We found that Linux's and QUIC's chromium implementation use Kathleen Nichols' algorithm to track windowed minimum-maximum values. We used the chromium implementation of windowed filter that uses C++ and made it compatible to work with ns-3. The source code for this work can be found here[1].

While these features are mandatory to implement TCP BBR in ns-3, they are generic features and can be leveraged by other CC algorithms. Hence, the implementation of the features mentioned above in ns-3 is not tightly coupled with that of TCP BBR model and can be used by other CC algorithms.

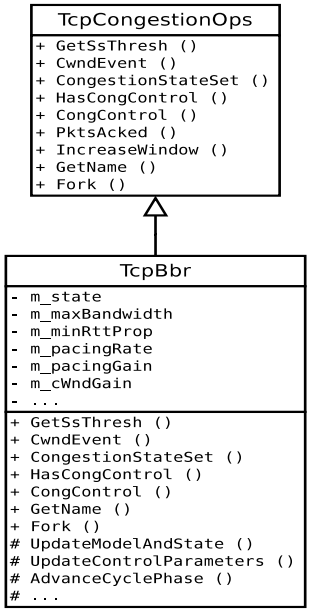
4 IMPLEMENTATION OF BBR MODEL IN NS-3

This section discusses the implementation of a new model for TCP BBR in ns-3. The proposed model is based on the guidelines provided in the Internet draft of TCP BBR [3]. The source code for same is under review and can be found here [2]. The functionality of TCP BBR is implemented in a new class called "`TcpBbr`" which is inherited from `TcpCongestionOps` (See Figure 2b). `TcpBbr` overrides

[1] https://goo.gl/18w8kY
[2] https://github.com/vivek-anand-jain/ns-3-dev-git/tree/bbr-dev



(a) TcpSocketBase interaction with TcpCongestionOps



(b) BBR class Diagram

Figure 2: BBR Implementation in ns-3

some of the methods of `TcpCongestionOps` and adds new methods that are specific to the functionality of TCP BBR. The following methods of `TcpCongestionOps` are overridden in `TcpBbr`:

(1) `uint32_t GetSshThresh ( )`: This method returns the current value of *ssthreshold*. In case of BBR, it returns `UINT32_MAX` because it does not consider the value of *ssthresh* during any phase of the connection.
(2) `void CwndEvent ( )`: This method is called to notify about various events on *cwnd*, like `CA_EVENT_COMPLETE_CWR`, `CA_EVENT_LOSS` and others. BBR uses this method to restore the *cwnd* when `CA_EVENT_COMPLETE_CWR` event occurs.
(3) `void CongestionSetState ( )`: This method is triggered on state change of a socket, e.g. `CA_OPEN`, `CA_RECOVERY`, `CA_LOSS`, `etc`. BBR uses this method to adjust its *cwnd* based on these states.
(4) `bool HasCongControl ( ) const`: This method returns true if CC algorithm is using `void CongControl ( )` instead of `void IncreaseWindow ( )`; otherwise false.
(5) `void CongControl ( )`: This method is used to update the BBR's network model and other control parameters. This method is called on receipt of every ACK.

Besides these methods, `TcpBbr` contains several methods that are specific to the functionality of TCP BBR. Some of the important ones are explained below; the rest are depicted in the call graph shown in Figure 3.

(1) `void UpdateModelAndState ( )`: This method is used to update BBR's network model (maximum bandwidth and minimum RTT) and other state information.
(2) `void UpdateControlParameters ( )`: This method is used to update control parameters i.e., *pacing rate, send quantum and cwnd.*
(3) `void EnterStartup ( ), void EnterDrain ( ), void EnterProbeBW ( ) void EnterProbeRTT ( )`: These methods initialize the state specific parameters.
(4) `void AdvanceCyclePhase ( )`: This method implements the functionality of gain cycling with randomization algorithm which is required during the ProbeBW state.

The interaction between `TcpSocketBase` and `TcpBbr` is shown in Figure 4. On receipt of every ACK, the state machine estimates the delivery rate and passes this information to `TcpBbr` via `CongControl ( )`. Based on this information, `TcpBbr` updates its network model and control parameters. On sending data, TCP socket checks the state of *pacing timer*. If *pacing timer* is running, TCP socket does not send any data and waits for it to expire. When the *pacing timer* is not running or expired, TCP socket directly sends the data and restarts the *pacing timer* based on current pacing rate and the amount of data sent.

Scope and Limitations

The proposed model for TCP BBR provides support for all the features suggested in the Internet Draft, except that the model does not support *send quantum* because it depends on segmentation offloading which is not available in ns-3.

5 MODEL EVALUATION

5.1 Model Verification

We have designed unit test cases to verify the implementation of TCP BBR model in ns-3. These tests are mandatory to ensure correctness of the proposed model before it can be merged into `ns-3-dev`. The following is a brief description of the test-suite comprising three high-level test cases:

- *Test 1*: This test verifies whether TCP socket is using pacing or not. The test should pass if the socket is using pacing. If pacing is not enabled, BBR must enable it. This test case is called two times: one with pacing enabled and another one

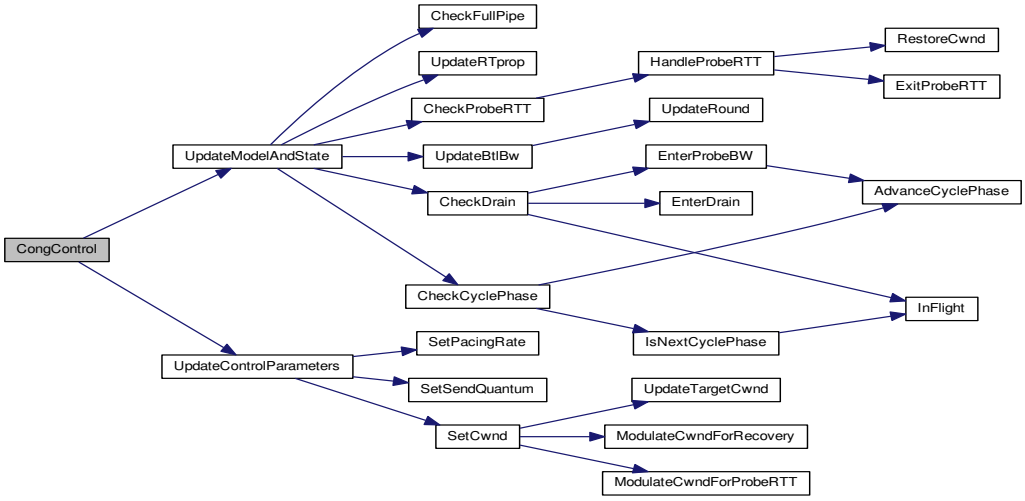


Figure 3: BBR call graph

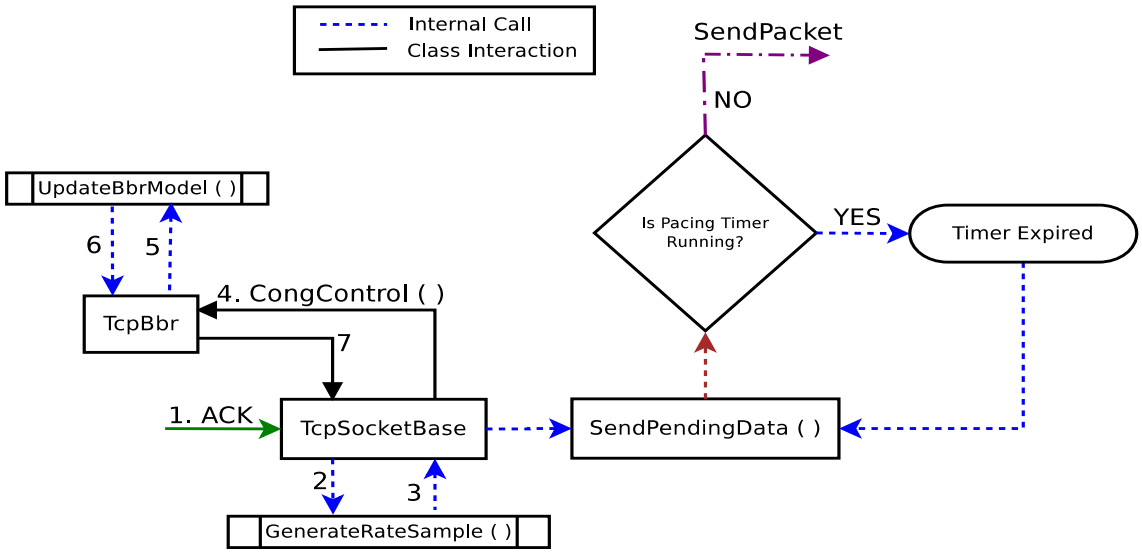


Figure 4: BBR Flow Diagram

with pacing disabled. BBR must not disable pacing in the former case and must enable pacing in the latter case.

- *Test 2*: This test verifies whether BBR appropriately sets the value of *cwnd_gain* and *pacing_gain* upon entering any state. For each state, it takes the high gain as input and compares the value of *cwnd_gain* and *pacing_gain* with the expected value.
- *Test 3*: This test verifies whether BBR correctly initializes the value of the internal variables based on initial *cwnd*. It accepts an arbitrary value for initial *cwnd*. This test case calls `CongestionStateSet` twice with `CA_OPEN` to verify whether BBR initializes the internal variables only once.

5.2 Functional Verification

In this section, we verify the correctness of our proposed model by testing the algorithm under different network conditions described in [2]. Our aim is to try and reproduce the results presented by the authors of BBR to confirm that our implementation exhibits key characteristics of BBR algorithm. Nonetheless, minor variations in the results are expected because: (i) the values of all the parameters used during the evaluation of BBR in [2] are not provided. We took

Table 1: Scenario Configuration

Parameter	Value
Topology	Dumbbell
TCP extension	BBR
Segment Size	536 Bytes
DelAckTimeout	200 ms
Initial Cwnd	10 segments
Limited Transmit	Enabled
Min RTO	200 ms
Window Scaling	Enabled
SACK	Enabled
Pacing	Enabled (Transport layer)
Bottleneck bandwidth	10 Mbps
Bottleneck buffer size	2 * BDP
Overall Propagation Delay	20 ms
Start Time	0.1 Seconds
End Time	100 Seconds

the values used in Linux for some parameters and for the rest (e.g., error rate in the network, background traffic, etc) we have made assumptions. (ii) segmentation offloading is a default feature in Linux, but it is not supported in ns-3.

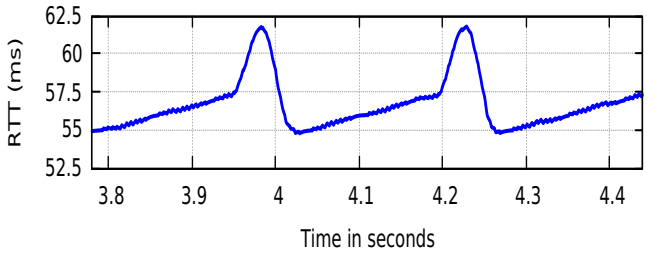
The simulation scenarios comprise of a dumbbell topology with one TCP BBR sender passing through the bottleneck bandwidth of 10 Mbps and bottleneck delay of 18 ms. Table 1 shows the values used for other simulation parameters. These values are used for all simulation scenarios unless otherwise specified. We evaluate the behavior of TCP BBR when the bottleneck bandwidth remains fixed (steady state), increases and decreases, and when the algorithm is in Startup and Drain phase. The results obtained from our model in ns-3 are compared to those presented by the authors of TCP BBR. For the ease of comparison, results presented by the authors of TCP BBR paper have been included in the Appendix section. The source code to reproduce these results has been made publicly available here[3].

5.2.1 Fixed Bottleneck Bandwidth. Figure 5 presents the results obtained for this scenario. We observe that the results obtained are identical to those presented in Figure 2 of [2]. This evaluation shows the impact of the *gain cycle* during the PROBE_BW phase when link bandwidth does not change. The triangular peaks are the effect of 1.25 and 0.75 pacing_gain.

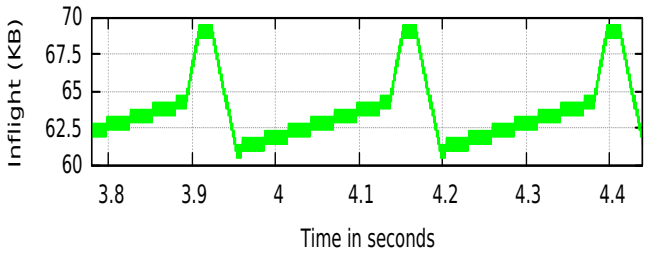
5.2.2 Increasing Bottleneck Bandwidth. Figure 3 of [2] depicts the behavior of BBR algorithm when the bandwidth is increased by two folds. It shows that BBR inflates its inflight data quickly to utilize the updated bandwidth without affecting the RTT. The periodic spikes are due to gain cycles. Our proposed model also depicts a similar behavior, as shown in Figure 6. The sudden fall in the amount of inflight data is due to a rapid increase in bandwidth which drains the queue

5.2.3 Decreasing Bottleneck Bandwidth. Figure 3 of [2] also shows the response of BBR algorithm when the bandwidth is reduced from 20 Mbps to 10 Mbps. It shows that BBR connection suffers from slightly higher RTT for few seconds because sudden

[3]https://github.com/Vivek-anand-jain/Reproduce-TCP-BBR-in-ns-3



(a) RTT (ms)



(b) Inflight data (KB).

Figure 5: Steady state behaviour

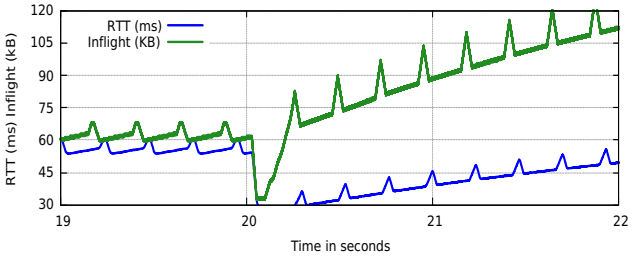


Figure 6: Increasing Bottleneck Bandwidth

decrement of bandwidth results into queue buildup. However, BBR adapts its network model based on the delivery rate estimation and reduces RTT to the minimum value. Figure 7 shows the results obtained from our model are in line with those presented in Figure 3 of [2].

5.2.4 Performance during Startup and Drain. BBR algorithm initiates in the STARTUP phase with the *cwnd_gain* and *pacing_gain* as $2/ln(2) \approx 2.89$. Within few RTTs, it fills the bottleneck queue causing bufferbloat. After achieving full bandwidth utilization, the algorithm enters into the DRAIN phase which results into drainage of the queue and clamps the growing RTT to the windowed minimum. Our simulation results, as shown in Figure 8, match the behavior depicted in Figure 4 of [2]. Since ns-3 does not have a built-in model for CUBIC [9] we have considered BIC for comparing the performance of BBR. Figure 8 shows that BBR initially grows faster than BIC. After realizing queue build up, BBR sender decreases its pacing rate to allow the queue to drain and clamps the growing RTT to the minimum value while for BIC, it keeps growing.

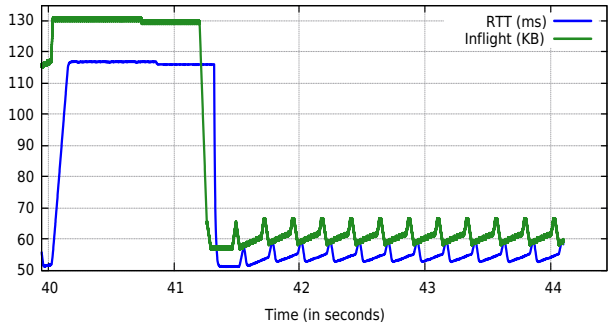


Figure 7: Decreasing Bottleneck Bandwidth

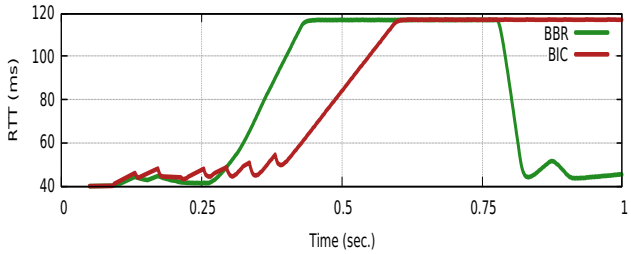


Figure 8: Performance during Startup and Drain

6 CONCLUSIONS

In this paper, we have discussed the implementation of pacing, delivery rate estimation and `cong_control` in ns-3 which are the prerequisite features for implementing TCP BBR. Subsequently, we have implemented a new model for TCP BBR in ns-3 and validated its functional characteristics. Our proposed model is under review with the ns-3 development team, and we aim to merge it in the native ns-3 distribution.

ACKNOWLEDGMENTS

We would like to thank Tom Henderson and Natale Patriciello for providing suggestions on our implementation.

REFERENCES

[1] M. Allman, H. Balakrishnan, and S. Floyd. Enhancing TCP's Loss Recovery Using Limited Transmit. RFC, United States, 2001.
[2] N. Cardwell, Y. Cheng, C. Gunn, S. Yeganeh, and V. Jacobson. BBR: Congestion-based Congestion Control. volume 14, pages 20–53, New York, NY, USA, 2016. ACM.
[3] N. Cardwell, Y. Cheng, S. Yeganeh, and V. Jacobson. BBR Congestion Control. Internet-Draft, 2017.
[4] N. Cardwell, Y. Cheng, S. Yeganeh, and V. Jacobson. Delivery Rate Estimation. Internet-Draft, 2017.
[5] M. Casoni and N. Patriciello. Next-generation TCP for ns-3 Simulator. volume 66, pages 81–93. Elsevier, 2016.
[6] E. Dumazet. TCP: Internal Implementation for Pacing, 2017.
[7] K. Fall and S. Floyd. Simulation-based Comparisons of Tahoe, Reno and SACK TCP. volume 26, pages 5–21. ACM, 1996.
[8] K. Fall and K. Varadhan. The Network Simulator (ns-2). *URL: http://www.isi.edu/nsnam/ns*, 2007.
[9] S. Ha, I. Rhee, and L. Xu. CUBIC: A New TCP-Friendly High-Speed TCP Variant. volume 42, pages 64–74. ACM, 2008.
[10] T. Henderson, M. Lacage, G. Riley, C. Dowell, and J. Kopena. Network Simulations with the ns-3 Simulator. volume 14, pages 527–527, 2008.
[11] V. Jacobson, R. Braden, and D. Borman. TCP Extensions for High Performance. 1992.
[12] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. TCP Selective Acknowledgment Options. Technical report, 1996.
[13] R. Stevens. TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms. 1997.