Some Deficiencies of TCP BBR Implementation in Ns-3 Network Simulator Detected under High Speed and Heavy Load Conditions

Toshihiko Kato†, Takahiko Kato‡, Ryo Yamamoto†, and Satoshi Ohzahata†

† University of Electro-Communications
Tokyo, Japan

‡ University of Fukui
Fukui, Japan

e-mail: kato@net.lab.uec.ac.jp, t-kato@u-fukui.ac.jp, ryo-yamamoto@uec.ac.jp, ohzahata@uec.ac.jp

***Abstract*— TCP Bottleneck Bandwidth and Round-trip propagation time (BBR) is a congestion control algorithm that was introduced by Google relatively recently, and it is widely used currently. BBR is based on the congestion-based congestion control which is different from the conventional algorithms. So, a lot of researches have been reported on the performance evaluation. In the early stage, it is done using the BBR software in the Linux operating system over a physical network. Nowadays, a network simulator, such as ns-3 is used for the evaluation. BBR was officially introduced in ns-3 version 3.34 released in 2021. However, the BBR software in ns-3 is different from that in Linux, and so it may be possible it behaves differently. This paper reports some results of BBR performance evaluation where sixteen flows share one 1 Gbps bottleneck link with a limited router buffer. The results were different when we used ns-3 versions 3.40 and 3.44. We also found some problems that the BBR state transition does not work correctly over a high speed link. We analyzed the behaviors of BBR program implemented in the ns-3 network simulator and examined the TCP BBR source code to find the reasons for those deficiencies. This paper describes the results of those investigations.**

Keywords- TCP; Congestion Control; BBR; Ns-3 Network Simulator.

I. INTRODUCTION

This paper is an extension of our previous paper [1], which was presented at the IARIA conference named AFIN 2025.

Along with the proliferation of various types of networks and applications, a lot of TCP variants with different congestion control algorithms are designed, implemented, and widely spread [2]. The congestion control manages the congestion window size, called cwnd according to a network congestion situation. The conventional algorithms are categorized into the loss-based, the delay-based and the hybrid approaches. The loss-based algorithm detects a network congestion by packet losses, and this category includes TCP NewReno [3], HighSpeed TCP [4], CUBIC TCP [5], and Hamilton TCP [6]. The delay-based algorithm detects a congestion by an increase of the Round-Trip Time (RTT), and an example of this category is TCP Vegas [7]. The hybrid algorithm combines the loss-based and delay-based approaches. In TCP Veno [8], the congestion is detected by packet losses and the change of cwnd is controlled according to RTT. In Compound TCP [9], cwnd is the sum of a window size following HighSpeed TCP and one following TCP Vegas, and it behaves in a way such that the former is effective when RTT is small and vice versa.

TCP BBR (Bottleneck Bandwidth and Round-trip propagation time) [10] was recently proposed by Google. It is a new type of congestion control algorithm called congestion-based congestion control. A data sender tries to send data segments according to the Bandwidth Delay Product (BDP) of the TCP connection. A sender measures the bottleneck bandwidth and the minimum RTT independently and estimates the BDP. It does not take care of retransmissions for the congestion control. Another feature of TCP BBR is that it follows the rate-based data transmission. Most of TCP variants adopt the window-based data transmission, where a sender sends out data segments in a burst within the limitation of window size (congestion window and advertised window). Instead, TCP BBR sends data segments in the pace determined by the estimated BDP and the segment size. As a result, data segments are transmitted sparsely and it is highly possible to avoid congestions in a multiple TCP flow condition.

TCP BBR came to be used widely since its proposal. Sawada et al. reported that TCP BBR was used by 30% of the major web servers in 2020. There are a lot of study results on the performance of TCP BBR [11]-[18]. In early stages, the performance evaluation was done using the BBR implementation in the Linux kernel over a physical network or a network emulator, such as Mininet [19]. Recently, TCP BBR was introduced in the network simulator, such as ns-3 [20]. Actually, the TCP BBR was implemented in ns-3 at version 3.34 released in 2021. Some studies performed experiments using ns-3 [15][17][18]. However, TCP BBR in ns-3 is implemented by its original source code, and so it may be possible that the code behaves differently from that in Linux.

We performed the TCP BBR throughput test under a high speed and heavy load condition where sixteen BBR flows share a 1Gbps bottleneck link [1]. When the output buffer of the router connected to the bottleneck link is limited, the throughput of some BBR flows becomes extremely low. Although Hock et al. reported the intra-protocol unfairness of TCP BBR due to small bottleneck buffer [11], the results of our experiment are much worse than that. We conducted this experiment using ns-3 whose version is 3.40. We performed the same experiment using ns-3 version 3.44, which was the newest as of writing our previous paper. The experimental results were different from that using ns-3 version 3.40, and all of the sixteen BBR

flows provided similar throughput. In this experiment, we also found that, in some cases, TCP BBR in ns-3 does not perform the BBR state transition for measuring the bandwidth and minimum RTT independently [21].

We analyzed the behaviors of TCP BBR in ns-3 in detail and consulted its BBR source code, from the two points of views, i.e. focusing on the difference of versions and on the BBR state transitions. This paper describes the results of our experiments and analysis of BBR behaviors.

The rest of this paper is organized as follows. Section II gives some background information including the overview of TCP BBR and the performance evaluation studies reported so far. Section III describes the details of the performance evaluation experiments focusing on the differences of ns-3 versions. Section IV compares the BBR codes implemented in ns-3 version 3.40 and 3.44 in detail. Section V discusses the detailed analysis focusing on the BBR state transitions. In the end, Section VI concludes this paper.

II. BACKGROUNDS

A. Overview of TCP BBR

TCP BBR uses the estimation for the available bottleneck bandwidth, *BtlBw*, and the minimal round-trip propagation time, *RTprop*, to calculate a path's available BDP. A BBR sends data segments with the pacing rate given by pacing_gain $\times$ BltBw $\times$ RTprop with the help of the rate-based control. The parameter pacing_gain, which is set to 1 most of the time, is used to control the actual pacing rate.

The BBR algorithm has four different states specific to BBR: Startup, Drain, Probe Bandwidth (ProbeBW), and Probe RTT (ProbeRTT).

The first phase starts in the Startup state in order to adapt the exponential startup behavior where the pacing_gain is set to 2/ln2 (2.885). Once the measured bandwidth does not increase further, BBR assumes to have reached the bottleneck bandwidth (maximum throughput), and shifts to the Drain state. Here, BBR reduces the pacing_gain to ln2/2.

Afterwards, BBR enters the ProbeBW state in which it probes for more available bandwidth. This is performed in eight cycles, each of which takes RTprop. The pacing_gain of each cycle is 5/4, 3/4, 1, 1, 1, 1, 1, and 1. In this state, BBR continuously samples the bandwidth and uses the maximum as the BtlBw estimator, whereby values are valid for the timespan of ten RTprop.

After not measuring a new RTprop value for time window W_R (typically, ten seconds), BBR stops probing for bandwidth and enters the ProbeRTT state. In this state, the bandwidth is reduced to four packets per sec. to drain any possible queue and get a real estimation of the RTT. This state is kept for 200 ms plus one RTT. If a new minimum value is measured, RTprop is updated and valid for W_R.

B. Studies on Performance Evaluation of TCP BBR

Table I summarizes the performance evaluation studies of TCP BBR. It shows the BBR source codes and the network environment used by the evaluation, the target TCP congestion control algorithms, and the maximum numbers of TCP flows examined in the experiment. TCP BBR was proposed in 2016, and introduced to the Linux operating system. So the evaluations in the early stage used the BBR code in Linux [11]-[14][16]. The BBR software was introduced to the ns-3 network simulator by Vivek Jain et al. [22] in 2018. This software was implemented over ns-3 version 3.27, which was released in 2017. References [15] and [17] used this version of TCP BBR. TCP BBR was introduced into ns-3 officially in the version 3.34 released in 2021. It is considered that [18] used this or later version of ns-3. The performance evaluation experiments were done for single BBR, multiple BBRs and mixture of multiple BBRs and other congestion control algorithms.

TABLE I. PERFORMANCE EVALUATION STUDIES OF TCP BBR.

reference	BBR code	network (link speed)	evaluation target	max number of flows
Hoch et al., 2017 [11]	Linux	physical (1, 10Gbps)	BBR, BBR + CUBIC	6 BBRs, 1 BBR + 1 CUBIC
Miyazawa et al., 2018 [12]	Linux	physical (1Gbps)	BBR + CUBIC	10 BBRs + 10 CUBICs
Sholz et al., 2018 [13]	Linux	Mininet (10Mbps)	BBR, BBR + CUBIC	6 BBRs, 10 BBRs + 10 CUBICs
Claypool et al., 2019 [14]	Linux	physical (40, 80, 120Mbps)	BBR, BBR + CUBIC	8 BBRs, 3 BBRs + 3 CUBICs
Zhang et al., 2019 [15]	ns-3 (Jain's version)	ns-3 (1, 10Gbps)	BBR, BBR + BIC, BBR + BIC + NewReno + Vegas	2 BBRs, 2 BBRs + 2 BIC, 1 BBR + 1 BIC + 1 NewReno + 1 Vegas
Kim et al., 2020 [16]	Linux	Mininet (100Mbps)	BBR, (Delay-aware BBR)	2 BBRs
Lanigan, 2020 [17]	ns-3 (Jain's version)	ns-3 (5, 10, 40Mbps)	BBR, (other BBR variants)	4 BBRs
Tang, 2024 [18]	ns-3	ns-3 (10, 100Mbps)	BBR, BBR + CUBIC	8 BBRs, 2 BBRs + 2 CUBICs
Ours, 2025 [1]	ns-3	ns-3 (1Gbps)	BBR	16 BBRs

As for the simulation studies shown by gray-colored boxes, the network bandwidth and the number of BBR flows are limited. Other than our result, only Zhang et al. use a Gbps level bandwidth, but this result evaluates the BBR performance only 10 sec., which is not enough to analyze the detailed BBR behaviors. The number of flows were limited, eight BBR flows at the maximum by Tang. In contrast with them, our performance evaluation used sixteen BBR flows using 1 Gbps links.

III. PERFORMANCE EVALUATION FOCUSING ON NS-3 VERSIONS

A. Experimental Setup

Figure 1 shows the network configuration used in this evaluation, which is a dumbbell configuration where sixteen data senders (S1 through S16) and the corresponding receivers (R1 through R16) are connected through two routers (N1 and N2). The bandwidth and transmission delay

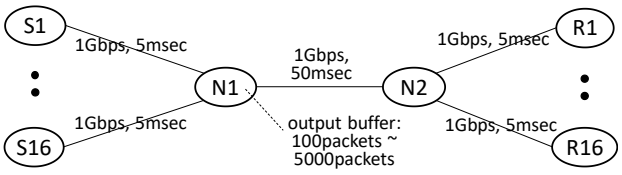


Figure 1. Performance evaluation network.

are specified in the figure, where the backbone link between N1 and N2 has a longer delay, 50 msec, than the other links. The output buffer in N1 is set to 100 packets through 5,000 packets. The maximum segment size is 1,420 bytes. Since RTT is 120 msec and the bottleneck bandwidth is 1 Gbps, the BDP in this experiment is 15 MB, i.e., around 10,000 packets.

The details of ns-3 parameters are as follows.

- Queue discipline: First-In First-Out queue discipline following the drop tail policy [23].
- TCP loss recovery algorithm: TCP classic recovery [24].
- DelAckCount (Number of packets to wait before sending a TCP Ack): 1.
- Explicit congestion notification functionality: not used.
- SACK: used.

In the evaluation, all data senders start data transfer at the same time and continue the communication until time 50 sec. We used the ns-3 version 3.40, released on Sep. 27, 2023, and version 3.44 released Mar. 9, 2025.

B. Evaluation Results Using Ns-3 Version 3.40

This subsection describes the results using ns-3 version 3.40. In the configuration shown in Figure 1, we set the output buffer of N1 to 100, 500, 1,000, 2,000, and 5,000 packets. As described above, the BDP in this configuration is around 10,000 packets, and so the buffer is not large enough, especially the value of 100 packets is extremely small.

Figure 2 shows the throughput of individual TCP BBR flows for five cases of the N1 output buffer. The bars are sorted in the descending order of throughput.

When the N1 output buffer is 100 packets, ten BBR flows provided high throughput more than 50 Mbps, and on the other hand, the other six flows provided very low throughput less than 1 Mbps. When the N1 output buffer is 500 and 1,000 packets, the results were similar. Nine or ten BBR flows out of the sixteen provided high throughput but the throughput of the rest was extremely low. But, for the cases that the N1 output buffer is 2,000 and 5,000 packets, the results were different, and almost all BBR flows provided high performance. Figure 3 shows the box-and-whisker plots where the whiskers show the minimum and maximum values, and the boxes give the first quartile, median and third quartile. The cross marks in the figure give the average. Table II shows the average, standard deviation, and total of sixteen flows' throughput. For the cases that the N1 output buffer is 100, 500, and 1,000 packets, the distribution of the throughput is large, and the total throughput is around 600 Mbps. Since the transmission rate of the bottleneck link is 1 Gbps, the efficiency is around 60 %. In contrast, in the cases that the N1 output buffer is 2,000 and 5,000 packets, the distribution becomes small and the efficiency is as high as 85 % or 90%.

Figure 4 shows the time variation of the pacing rate of the 6th BBR flow (flow 6) and the 12th BBR flow (flow 12) when the N1 output buffer is 100 packets. In the beginning, the pacing rate increases similarly and some packet losses following timeout retransmissions. After that, the pacing rate of flow 12 keeps a very low value, such as 0.1 and 0.01 Mbps. In contrast, that of 6th flow recovers to the value of 87 Mbps. This situation continues to the end of experiment. Other flows showed similar behaviors. This is the reason for the high and low throughput.

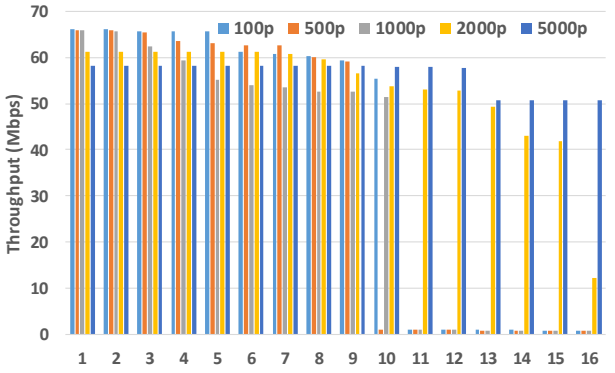


Figure 2. Throughput of individual BBR flows.

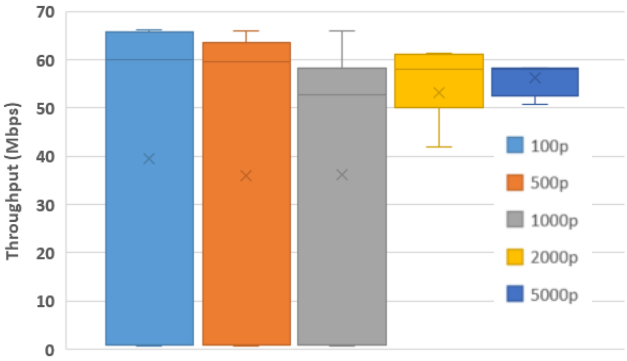


Figure 3. Box-and-whisker plots of BBR flow throughput.

TABLE II. AVERAGE, STANDARD DEVIATION, AND TOTAL OF THROUGHPUT OF BBR FLOWS (MBPS).

buffer size	average	std. dev.	total
100 pkts.	39.5	31.0	632.0
500 pkts.	35.9	32.0	574.8
1000 pkts.	36.1	28.5	577.9
2000 pkts.	53.1	12.7	850.3
5000 pkts.	56.3	3.3	900.2

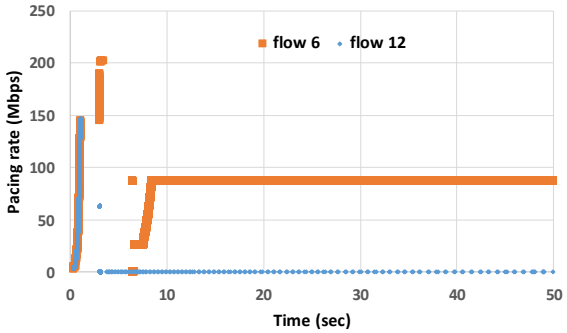


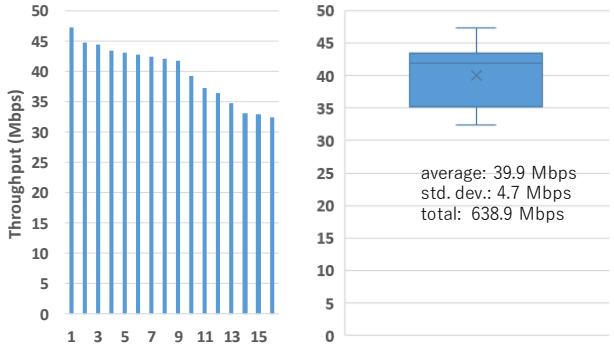
Figure 4. Pacing rate vs. time for flow 6 and 12 when the N1 output buffer is 100 packets.

C. Evaluation Results Using Ns-3 Version 3.44

Next, we performed the same experiment using ns-3 version 3.44. Actually, we used the same script file as that for ns-3 version 3.40. Figure 5 shows the results for the case that the N1 output buffer is 100 packets. Figure 5 (a) gives the individual throughput of sixteen BBR flows and Figure 5 (b) is its box-and-whisker plot. The distribution becomes small and all flows provide high performance, between 32 Mbps and 47 Mbps. It should be noted that the total throughput is similar to that in ns-3 version 3.40. This means that the behaviors of flows is fair compared with the former experiment.

Figure 6 shows the time variation of the pacing rate of flow 5. The behavior in the first 10 seconds is similar to that of ns-3 version 3.40 depicted in Figure 4. After that, however, the pacing rate changes among 50 Mbps, 70 Mbps, and 85 Mbps. For other flows, the behavior of pacing rate is similar, and as a result, the throughput of individual flows provided similar value in the experiment using ns-3 version 3.44.

Both results seem to be explainable. As mentioned above, it is possible that TCP BBR shows the intra-protocol unfairness when the bottleneck buffer is limited [11]. So, the results by using ns-3 version 3.40 are not necessarily wrong. The problem is that the behavior changed for ns-3 version 3.40 and 3.44. The next section discusses this issue in more detail.



(a) Individual throughput (b) Box-and-whisker plots

Figure 5. Results by ns-3 3.44 for 100 packet N1 output buffer.

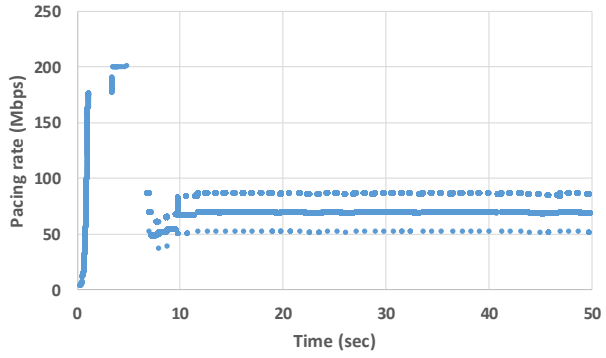


Figure 6. Pacing rate vs. time for flow 5 by ns-3 3.44 when the N1 output buffer is 100 packets.

IV. Detailed Analysis of Ns-3 BBR Source Code Focusing on Versions

A. Program Strucsture of Ns-3 BBR Code

In ns-3, the main procedures of TCP are implemented in the tcp-socket-base.{h, cc} files. They give the functions for handling data send requests from the upper layer and for handling TCP segments received from the remote node. The control variables, such as `m_nextTxSequence` and `m_cWnd` are implemented in tcp-socket-state.h. The congestion control algorithms are implemented in other files independently. TCP BBR is implemented in the tcp-bbr.{h, cc} files.

Figure 7 depicts the function call graph of the TCP BBR congestion control (function `CongControl()`). It calls two main functions, `UpdateModelAndState()` and `UpdateControlParameters()`. The former calls the functions changing the states corresponding to BtlBw and RTprop. The latter calls the functions updating the related parameters, such as `m_pacingRate` and `m_cWnd`. The overviews of those functions are listed in Table III.

Figure 8 depicts the function call graph in tcp-socket-base.cc when an ACK segment is received. First, `ReceivedAck()` is called. As for the ACK processing, it calls `ProcessAck()`, and `CongControl()` in tcp-bbr.cc. `ProcessAck()` manages the ACK related parameters, and if the received ACK is a duplicate ACK, `DupAck()` is called and the retransmission is processed if necessary. After `CongControl()`, `SendPendingData()` is called to send the following data when the received ACK opens a window. In BBR, the rate-based data transfer is adopted and this is implemented by the TCP Pacing mechanism [25]. A timer called `m_pacingTimer` is used, which expires according to a rate to send data segments according to `m_pacingRate`, calculated in `SetPacingRate()`.

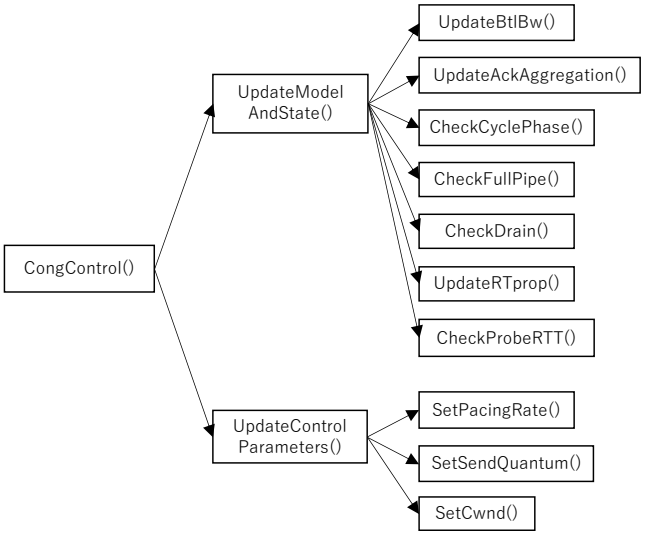


Figure 7. Function calls in TCP BBR Code in ns-3 version 3.40.

TABLE III. OVERVIEW OF BBR FUNCTIONS IN FIG. 7.

function	behavior
UpdateBtlBw()	Updates maxBwFilter if a new delivery rate is larger than prior value.
UpdateAck Aggregation()	Takes account of delayed ACK.
CheckCyclePhase()	If in ProbeBW phase and a RTT elapsed, advance cycle.
CheckFullPipe()	If delivery rate becomes large enough, set m_isPipeFilled = true
CheckDrain()	If in Startup phase and m_isPipeFilled is true, enter Drain phase. If in Drain phase and inflight data decreased, enter ProbeBW phase.
UpdateRTprop()	If RTT didn't increase or W_R (10 sec) elapsed (m_rtPropExpired = true), update m_rtProp.
CheckProbeRTT()	If not in ProbeRTT phase and m_rtPropExpierd, enter ProbeRTT phase and save cwnd. If in ProbeRTT phase and 200 msec elapsed, enter ProbeBW if m_isPipeFilled or Startup otherwise.
SetPacingRate()	Set rate = Max value in maxBwFilter × pacing_gain. If m_isPipeFilled or rate > m_pacingRate, set m_pacingRate = rate.
SetSendQuantum()	Set m_sendQuantum = MSS.
SetCwnd()	Determine cwnd considering targetCwnd and packet losses. If in ProbeRTT phase, set cwnd to 4 MSS.

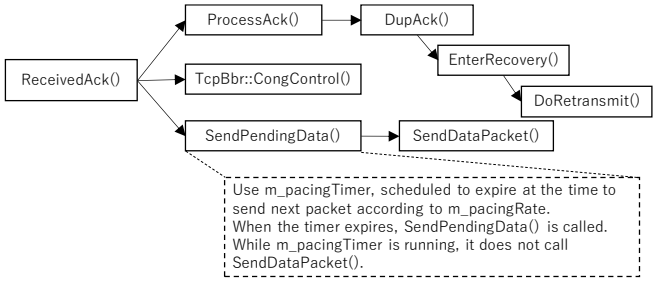


Figure 8. Function calls in TCP when receiving ACK in ns-3 version 3.40.

These are described based on the program structure of ns-3 version 3.40. The structure itself is the same in version 3.44.

B. Analysis on Difference of Ns-3 3.40 and 3.44

We analyzed the BBR codes in version 3.40 and 3.44 to clarify why the results are different in these two versions. From the homepage of ns-3 [26], the TCP related changes from version 3.40 to 3.44 are as follows.

1) TCP Cubic now supports TCP-friendliness by default, making the congestion window growth somewhat more aggressive. (from 3.40 to 3.41)
2) TcpCubic and TcpLinuxReno will no longer grow their cwnd when application-limited. (from 3.41 to 3.42)
3) Deprecated EventId::IsRunning(). It has been replaced with EventId::IsPending(). (from 3.41 to 3.42)
4) TCP Proportional Rate Reduction (PRR) recovery has been aligned to the updates in draft-ietf-tcpm-prr-rfc6937bis. (from 3.42 to 3.43)
5) A new trace source TcpSocketBase::LastRtt has been added for tracing the last RTT sample observed. The existing trace source TcpSocketBase::Rtt is still providing the smoothed RTT, although it had been incorrectly documented as providing the last RTT. (from 3.42 to 3.43)

Among them, 1), 2), and 4) do not give any impacts to our experiment, because we do not use TCP Cubic, NewReno, nor PRR recovery.

Then we compared the source codes of tcp-bbr.{h, cc} and tcp-socket-base.{h, cc}. The followings are differences that are considered to give some impacts to the BBR behaviors.

tcp-bbr.cc:

- `m_rtProp` in 3.40 is changed to `m_minRtt` in 3.44. This seems to be just a text-level modification.
- After rate is calculated in `SetPacingRate()` as shown in Table III, the following modification is added.
  `rate *= (1.f – m_pacingMargin),`
  `m_pacingMargin` is set to 0.01
  This needs to be considered.
- One `if` sentence is modified in `UpdateBltBw()`.
  3.40: `if (rs.m_deliveryRate == 0)`
  3.44: `if (rs.m_delivered < 0 || interval.IsZero())`
  This needs to be considered.
- `m_rtProp` (`m_minRtt`) is set to `m_lastRtt` in 3.40, but to `m_srtt` in 3.44. This is related to item 5) in the change history. This may need to be considered.

tcp-socket-base.cc

- One else if sentence is modified in `DupAck()`:
  3.40: `else if (m_txBuffer->IsLost(m_highRxAckMark + m_tcb->m_segmentSize)`
  3.44: `else if (m_txBuffer->IsLost(m_highRxAckMark)`
  This needs to be considered.
- Related to item 5) in the change history, `EstimateRtt()` is largely modified. This needs to be considered.

Based on these considerations, we took the following two approaches.

Approach 1: Modify tcp-bbr.cc and tcp-socket-base.cc for the points we decided to be considered.

Approach 2: Use tcp-bbr.cc and tcp-socket-base.cc of version 3.44 in ns-3 version 3.44.

Approach 2 required the following modification in the BBR source code,

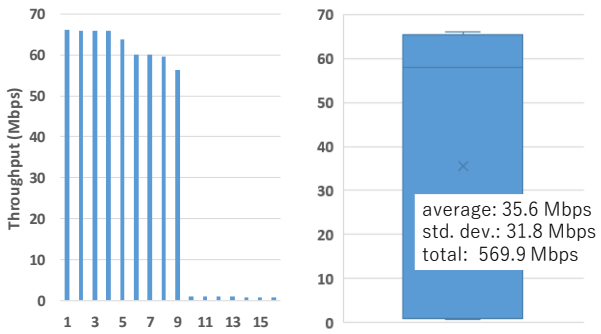
tcp-socket-base.cc to be ported:

- Change `IsPending()` for `EventId` variables back to `IsRunning()`. This is related to item 3) in the change history.

tcp-socket-state.h in version 3.40:

- Define `TraceValue<Time> m_srtt` and `bool m_isCwndLimited`.

We performed both examination for the case that the bottleneck buffer is 100 packet. The results were similar and did not improve the performance of ns-3 version 3.40. Figure 9 shows the result of Approach 2. Out of the sixteen BBR flows, nine provided high throughput but the throughput of the other seven flows was extremely low. The average and total throughput was even worse than that of the original version 3.40. This result says that the difference of BBR behaviors in ns-3 versions 3.40 and 3.44 may come from some other parts in the simulator. In the current stage, we do not clarify the details.



(a) Individual throughput (b) Box-and-whisker plots

Figure 9. Results of Approach 2 porting 3.44 BBR code to 3.40 for 100 packet N1 output buffer.

V. Detailed Analysis of Focusing on BBR State Transitions

A. State Transition in Ns-3 BBR Implementation

As stated in Section II, TCP BBR introduces its own state transition to estimate BtlBw and RTprop independently. After the initial state transition from Startup and Drain states, BBR repeatedly takes states ProbeBW and ProbeRTT, each of which is designed for estimating BltBw and RTprop, respectively. Especially, BBR shifts its state to ProbeRTT by a time window basis of W_R, whose value is 10 sec. by default.

Figure 10 shows the state transition log of BBR flows executed by ns-3 version 3.44. In each flow, the state starts from Startup and shifts to Drain. Although the state is changed to ProbeBW immediately, there are no state changes after that. That means there are no state transitions to ProbeRTT. Figure 10 shows the results of five flows. The other even flows showed the similar state transitions. In the case of ns-3 version 3.40, the results were the same.

B. Analysis oc Ns-3 BBR Source Code Focusing on State Transitions

Among the functions listed in Table III, the BBR state transition from ProbeBW to ProbeRTT is performed in function `CheckProbeRTT()`. Figure 11 shows the corresponding source code. The state transition itself is performed in function `EnterProbeRTT()` in the figure, and the condition to execute it is that the state is not ProbeRTT, `m_minRttExpired` is true, and `m_idleRestart` is false. The reason for not shifting to ProbeRTT is supposed that `m_minRttExpired` is always false, or that `m_idleRestart` is always true. By examining the values of `m_minRttExpired` and `m_idleRestart` in the experiment using version 3.44 BBR, we found that `m_minRttExpired` is always false in the experiment.

```
flow 1 TcpBbr:SetBbrState(): At 0.12 Set to BBR_STARTUP
flow 1 TcpBbr:SetBbrState(): At 3.6997 Changing from BBR_STARTUP to BBR_DRAIN
flow 1 TcpBbr:SetBbrState(): At 3.6997 Changing from BBR_DRAIN to BBR_PROBE_BW

flow 2 TcpBbr:SetBbrState(): At 0.12 Set to BBR_STARTUP
flow 2 TcpBbr:SetBbrState(): At 3.5903 Changing from BBR_STARTUP to BBR_DRAIN
flow 2 TcpBbr:SetBbrState(): At 3.5983 Changing from BBR_DRAIN to BBR_PROBE_BW

flow 3 TcpBbr:SetBbrState(): At 0.12 Set to BBR_STARTUP
flow 3 TcpBbr:SetBbrState(): At 3.5799 Changing from BBR_STARTUP to BBR_DRAIN
flow 3 TcpBbr:SetBbrState(): At 3.5799 Changing from BBR_DRAIN to BBR_PROBE_BW

flow 4 TcpBbr:SetBbrState(): At 0.12 Set to BBR_STARTUP
flow 4 TcpBbr:SetBbrState(): At 3.5813 Changing from BBR_STARTUP to BBR_DRAIN
flow 4 TcpBbr:SetBbrState(): At 3.5895 Changing from BBR_DRAIN to BBR_PROBE_BW
. . .
flow 16 TcpBbr:SetBbrState(): At 0.12 Set to BBR_STARTUP
flow 16 TcpBbr:SetBbrState(): At 3.4705 Changing from BBR_STARTUP to BBR_DRAIN
flow 16 TcpBbr:SetBbrState(): At 3.4705 Changing from BBR_DRAIN to BBR_PROBE_BW
```

Figure 10. BBR state transition log in version 3.44 BBR program.

Based on this result, we examined the source code for setting `m_minRttExpired`. This is performed in function `UpdateRTprop()` shown in Figure 12. In this function, `m_minRttExpired` is set to a boolean value whether the current time is larger than `m_minRttStamp` + W_R (10 sec.). Here, `m_minRttStamp` is the time when a new value was assigned to `m_minRtt`, and `m_minRtt` corresponds to RTprop.

The latter part of Figure 12 shows how to update `m_minRtt` and `m_minRttStamp`. Variable `tcb->m_lastRtt` used here indicates the RTT value of the last ACKed packet. The condition to update the variables is `tcb->m_lastRtt` is not greater than `m_minRtt`, or `m_minRttExpired` is true.

In the experiment, we obtained the program log shown in Figure 13 as for the update of `m_minRttStamp` for flow 1. As shown in the figure, these updates were performed 89,422 times during the fifty second program run. Ideally, `m_minRttStamp` is updated every ten second along with `m_minRttExpired` being true as described above. However, too many advances of `m_minRttStamp` make the sum of itself and W_R larger than the current time and keep `m_minRttExpired` false.

Since `m_minRttExpired` is always false, the condition that allows `m_minRttStamp` to be updated is supposed to be "`tcb->m_lastRtt <= m_minRtt`." We examined the values of `tcb->m_lastRtt` and

```
if (m_state != BbrMode_t::BBR_PROBE_RTT
   && m_minRttExpired && !m_idleRestart)
   {
      EnterProbeRTT();
      SaveCwnd(tcb);
      m_probeRttDoneStamp = Seconds(0);
   }
```

Figure 11. Process to enter ProbeRTT state in `CheckProbeRTT()`.

```
m_minRttExpired = Simulator::Now() >
        (m_minRttStamp + m_minRttFilterLen);

if (tcb->m_lastRtt.Get().IsPositive()
   && (tcb->m_lastRtt <= m_minRtt ||
       m_minRttExpired))
   {
      m_minRtt = tcb->m_lastRtt;
      m_minRttStamp = Simulator::Now();
   }
```

Figure 12. Process to update `m_minRttExpired` in `UpdateRTprop()`.

```
TcpBbr:UpdateRTprop(): At 0.048 minRttStamp is updated.
TcpBbr:UpdateRTprop(): At 0.0482 minRttStamp is updated.
TcpBbr:UpdateRTprop(): At 0.0484 minRttStamp is updated.
TcpBbr:UpdateRTprop(): At 0.0486 minRttStamp is updated.
TcpBbr:UpdateRTprop(): At 0.0488 minRttStamp is updated.
TcpBbr:UpdateRTprop(): At 0.049 minRttStamp is updated.
TcpBbr:UpdateRTprop(): At 0.0721 minRttStamp is updated.
TcpBbr:UpdateRTprop(): At 0.0729 minRttStamp is updated.
TcpBbr:UpdateRTprop(): At 0.0737 minRttStamp is updated.
TcpBbr:UpdateRTprop(): At 0.0746 minRttStamp is updated.
…
Total number is 89,442.
```

Figure 13. `m_minRttStamp` update log in `UpdateRTprop()` for.flow 1.

```
TcpBbr:UpdateRTprop(): At 0.048 tcb->m_lastRtt is 0.024 m_minRtt is 0.024
TcpBbr:UpdateRTprop(): At 0.0482 tcb->m_lastRtt is 0.024 m_minRtt is 0.024
TcpBbr:UpdateRTprop(): At 0.0484 tcb->m_lastRtt is 0.024 m_minRtt is 0.024
TcpBbr:UpdateRTprop(): At 0.0486 tcb->m_lastRtt is 0.024 m_minRtt is 0.024
TcpBbr:UpdateRTprop(): At 0.0488 tcb->m_lastRtt is 0.024 m_minRtt is 0.024
TcpBbr:UpdateRTprop(): At 0.049 tcb->m_lastRtt is 0.024 m_minRtt is 0.024
TcpBbr:UpdateRTprop(): At 0.0492 tcb->m_lastRtt is 0.025 m_minRtt is 0.024
TcpBbr:UpdateRTprop(): At 0.0494 tcb->m_lastRtt is 0.025 m_minRtt is 0.024
TcpBbr:UpdateRTprop(): At 0.0495 tcb->m_lastRtt is 0.025 m_minRtt is 0.024
TcpBbr:UpdateRTprop(): At 0.0497 tcb->m_lastRtt is 0.025 m_minRtt is 0.024
TcpBbr:UpdateRTprop(): At 0.0721 tcb->m_lastRtt is 0.024 m_minRtt is 0.024
TcpBbr:UpdateRTprop(): At 0.0729 tcb->m_lastRtt is 0.024 m_minRtt is 0.024
TcpBbr:UpdateRTprop(): At 0.0737 tcb->m_lastRtt is 0.024 m_minRtt is 0.024
TcpBbr:UpdateRTprop(): At 0.0746 tcb->m_lastRtt is 0.024 m_minRtt is 0.024
TcpBbr:UpdateRTprop(): At 0.0754 tcb->m_lastRtt is 0.024 m_minRtt is 0.024
m_lastRtt is 0.024 continuously …
TcpBbr:UpdateRTprop(): At 0.0862 tcb->m_lastRtt is 0.024 m_minRtt is 0.024
TcpBbr:UpdateRTprop(): At 0.087 tcb->m_lastRtt is 0.025 m_minRtt is 0.024
TcpBbr:UpdateRTprop(): At 0.0879 tcb->m_lastRtt is 0.024 m_minRtt is 0.024
TcpBbr:UpdateRTprop(): At 0.0961 tcb->m_lastRtt is 0.024 m_minRtt is 0.024
TcpBbr:UpdateRTprop(): At 0.0969 tcb->m_lastRtt is 0.024 m_minRtt is 0.024
TcpBbr:UpdateRTprop(): At 0.0978 tcb->m_lastRtt is 0.024 m_minRtt is 0.024
TcpBbr:UpdateRTprop(): At 0.0986 tcb->m_lastRtt is 0.024 m_minRtt is 0.024
TcpBbr:UpdateRTprop(): At 0.0994 tcb->m_lastRtt is 0.024 m_minRtt is 0.024
TcpBbr:UpdateRTprop(): At 0.1003 tcb->m_lastRtt is 0.024 m_minRtt is 0.024
TcpBbr:UpdateRTprop(): At 0.1011 tcb->m_lastRtt is 0.024 m_minRtt is 0.024
TcpBbr:UpdateRTprop(): At 0.1019 tcb->m_lastRtt is 0.024 m_minRtt is 0.024
TcpBbr:UpdateRTprop(): At 0.1028 tcb->m_lastRtt is 0.024 m_minRtt is 0.024
TcpBbr:UpdateRTprop(): At 0.1036 tcb->m_lastRtt is 0.024 m_minRtt is 0.024
TcpBbr:UpdateRTprop(): At 0.1044 tcb->m_lastRtt is 0.024 m_minRtt is 0.024
TcpBbr:UpdateRTprop(): At 0.1052 tcb->m_lastRtt is 0.024 m_minRtt is 0.024
TcpBbr:UpdateRTprop(): At 0.106 tcb->m_lastRtt is 0.025 m_minRtt is 0.024
TcpBbr:UpdateRTprop(): At 0.1068 tcb->m_lastRtt is 0.024 m_minRtt is 0.024
TcpBbr:UpdateRTprop(): At 0.1075 tcb->m_lastRtt is 0.024 m_minRtt is 0.024
TcpBbr:UpdateRTprop(): At 0.1082 tcb->m_lastRtt is 0.024 m_minRtt is 0.024
…
```

Figure 14. Log of `tcb->m_lastRtt` and `m_minRttStamp`.for flow 1.

`m_minRtt` when these variables are updated. Figure 14 gives a part of the results for flow 1. As shown by the numbers with the underlined and bold form in this figure, `tcb->m_lastRtt` and `m_minRtt` often take the same value. This introduces the frequent updates of `m_minRttStamp`.

We examined the BBR source code in the Linux operating system. Figure 15 shows the corresponding part of the BBR source code [27]. The ns-3 BBR source code has a similar structure with that in Linux. `UpdateRTprop()` in ns-3 corresponds to `bbr_update_min_rtt()` in Linux. `tcb->m_lastRtt` and `m_minRtt` correspond to `rs->rtt_us` and `bbr->probe_rtt_min_us`, respectively. So, the condition we focus on corresponds to

`rs->rtt_us < bbr->probe_rtt_min_us`,

```
static void bbr_update_min_rtt(struct sock *sk, const struct rate_sample *rs)
{
    expire = bbr->probe_rtt_min_stamp +
             msecs_to_jiffies(bbr_param(sk, probe_rtt_win_ms));
    probe_rtt_expired = after(tcp_jiffies32, expire);
    if (rs->rtt_us >= 0 && (rs->rtt_us < bbr->probe_rtt_min_us ||
                            (probe_rtt_expired && !rs->is_ack_delayed))) {
        bbr->probe_rtt_min_us = rs->rtt_us;
        bbr->probe_rtt_min_stamp = tcp_jiffies32;
    }
```

Figure 15. Corresponding source code in Linux.

highlighted in the figure by the underlined and bold style. From these analysis, it is obvious that the condition we focus on should be

`tcb->m_lastRtt < m_minRtt`,

instead of

`tcb->m_lastRtt <= m_minRtt`.

The discussions above were done for the ns-3 version 3.44 BBR program. The situation was the same for the ns-3 version 3.40 BBR program.

C. Results of Bug Fixing for Ns-3 Ver.3.44 BBR Program

We fixed the bug described above in the version 3.44 BBR program.

The throughput of individual flows is as shown in Figure 16. By comparing this result and that given in Figure 5, we can conclude that the throughput has the similar tendency for the original 3.44 BBR program and the bug-fixed 3.44 BBR program.

Figure 17 shows the BBR state transition logs for the bug-fixed program. Comparing this figure with Figure 10, we can conclude that the BBR state transition is performed well in this result. The state transitions from ProbeBW to ProbeRTT are performed repeatedly.

But there is another problem in this result. It is considered that the state transitions to ProbeRTT need to be observed four times in each flow. However, the count of these transitions in the experiment is less than four. For example, only one transition to ProbeRTT occurs in flows 2 and 3. In flows 1 and 9, the count of the transitions to Probe RTT is two, and in flows 4, 15, and 16 the count is three. Actually, there were no flows with four time transitions among sixteen flows in the experiment.

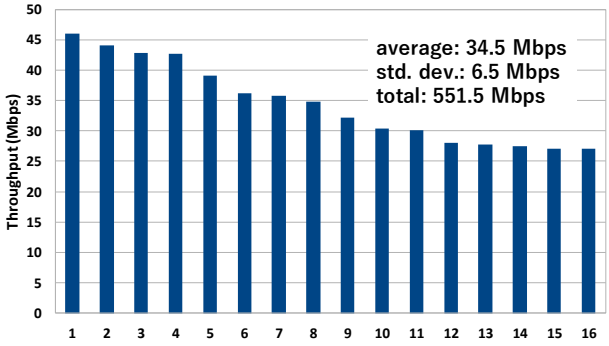


Figure 16. Throughput of individual flows using bug-fixed 3.44 BBR program for 100 packet N1 output buffer.

```
flow 1 TcpBbr:SetBbrState(): At 0.024 Set to BBR_STARTUP
flow 1 TcpBbr:SetBbrState(): At 1.1993 Changing from BBR_STARTUP to BBR_DRAIN
flow 1 TcpBbr:SetBbrState(): At 1.1993 Changing from BBR_DRAIN to BBR_PROBE_BW
flow 1 TcpBbr:SetBbrState(): At 11.6166 Changing from BBR_PROBE_BW to BBR_PROBE_RTT
flow 1 TcpBbr:SetBbrState(): At 11.8413 Changing from BBR_PROBE_RTT to BBR_PROBE_BW
flow 1 TcpBbr:SetBbrState(): At 21.8414 Changing from BBR_PROBE_BW to BBR_PROBE_RTT
flow 1 TcpBbr:SetBbrState(): At 22.8356 Changing from BBR_PROBE_RTT to BBR_PROBE_BW

flow 2 TcpBbr:SetBbrState(): At 0.024 Set to BBR_STARTUP
flow 2 TcpBbr:SetBbrState(): At 0.254 Changing from BBR_STARTUP to BBR_DRAIN
flow 2 TcpBbr:SetBbrState(): At 1.2264 Changing from BBR_DRAIN to BBR_PROBE_BW
flow 2 TcpBbr:SetBbrState(): At 10.639 Changing from BBR_PROBE_BW to BBR_PROBE_RTT
flow 2 TcpBbr:SetBbrState(): At 10.8553 Changing from BBR_PROBE_RTT to BBR_PROBE_BW

flow 3 TcpBbr:SetBbrState(): At 0.024 Set to BBR_STARTUP
flow 3 TcpBbr:SetBbrState(): At 0.2524 Changing from BBR_STARTUP to BBR_DRAIN
flow 3 TcpBbr:SetBbrState(): At 1.2264 Changing from BBR_DRAIN to BBR_PROBE_BW
flow 3 TcpBbr:SetBbrState(): At 10.8866 Changing from BBR_PROBE_BW to BBR_PROBE_RTT
flow 3 TcpBbr:SetBbrState(): At 11.1029 Changing from BBR_PROBE_RTT to BBR_PROBE_BW

flow 4 TcpBbr:SetBbrState(): At 0.024 Set to BBR_STARTUP
flow 4 TcpBbr:SetBbrState(): At 0.246 Changing from BBR_STARTUP to BBR_DRAIN
flow 4 TcpBbr:SetBbrState(): At 0.2673 Changing from BBR_DRAIN to BBR_PROBE_BW
flow 4 TcpBbr:SetBbrState(): At 10.8754 Changing from BBR_PROBE_BW to BBR_PROBE_RTT
flow 4 TcpBbr:SetBbrState(): At 11.0918 Changing from BBR_PROBE_RTT to BBR_PROBE_BW
flow 4 TcpBbr:SetBbrState(): At 21.092 Changing from BBR_PROBE_BW to BBR_PROBE_RTT
flow 4 TcpBbr:SetBbrState(): At 21.3396 Changing from BBR_PROBE_RTT to BBR_PROBE_BW
flow 4 TcpBbr:SetBbrState(): At 32.0601 Changing from BBR_PROBE_BW to BBR_PROBE_RTT
flow 4 TcpBbr:SetBbrState(): At 32.2808 Changing from BBR_PROBE_RTT to BBR_PROBE_BW
...
flow 9 TcpBbr:SetBbrState(): At 0.024 Set to BBR_STARTUP
flow 9 TcpBbr:SetBbrState(): At 0.2533 Changing from BBR_STARTUP to BBR_DRAIN
flow 9 TcpBbr:SetBbrState(): At 1.2302 Changing from BBR_DRAIN to BBR_PROBE_BW
flow 9 TcpBbr:SetBbrState(): At 10.6391 Changing from BBR_PROBE_BW to BBR_PROBE_RTT
flow 9 TcpBbr:SetBbrState(): At 10.8554 Changing from BBR_PROBE_RTT to BBR_PROBE_BW
flow 9 TcpBbr:SetBbrState(): At 31.0323 Changing from BBR_PROBE_BW to BBR_PROBE_RTT
flow 9 TcpBbr:SetBbrState(): At 31.9589 Changing from BBR_PROBE_RTT to BBR_PROBE_BW
...
flow 15 TcpBbr:SetBbrState(): At 0.024 Set to BBR_STARTUP
flow 15 TcpBbr:SetBbrState(): At 0.2469 Changing from BBR_STARTUP to BBR_DRAIN
flow 15 TcpBbr:SetBbrState(): At 1.2267 Changing from BBR_DRAIN to BBR_PROBE_BW
flow 15 TcpBbr:SetBbrState(): At 10.0241 Changing from BBR_PROBE_BW to BBR_PROBE_RTT
flow 15 TcpBbr:SetBbrState(): At 10.264 Changing from BBR_PROBE_RTT to BBR_PROBE_BW
flow 15 TcpBbr:SetBbrState(): At 20.2643 Changing from BBR_PROBE_BW to BBR_PROBE_RTT
flow 15 TcpBbr:SetBbrState(): At 20.5036 Changing from BBR_PROBE_RTT to BBR_PROBE_BW
flow 15 TcpBbr:SetBbrState(): At 41.262 Changing from BBR_PROBE_BW to BBR_PROBE_RTT
flow 15 TcpBbr:SetBbrState(): At 41.472 Changing from BBR_PROBE_RTT to BBR_PROBE_BW

flow 16 TcpBbr:SetBbrState(): At 0.024 Set to BBR_STARTUP
flow 16 TcpBbr:SetBbrState(): At 1.1993 Changing from BBR_STARTUP to BBR_DRAIN
flow 16 TcpBbr:SetBbrState(): At 1.1993 Changing from BBR_DRAIN to BBR_PROBE_BW
flow 16 TcpBbr:SetBbrState(): At 10.7939 Changing from BBR_PROBE_BW to BBR_PROBE_RTT
flow 16 TcpBbr:SetBbrState(): At 11.0102 Changing from BBR_PROBE_RTT to BBR_PROBE_BW
flow 16 TcpBbr:SetBbrState(): At 21.0102 Changing from BBR_PROBE_BW to BBR_PROBE_RTT
flow 16 TcpBbr:SetBbrState(): At 21.957 Changing from BBR_PROBE_RTT to BBR_PROBE_BW
flow 16 TcpBbr:SetBbrState(): At 32.0349 Changing from BBR_PROBE_BW to BBR_PROBE_RTT
flow 16 TcpBbr:SetBbrState(): At 33.0316 Changing from BBR_PROBE_RTT to BBR_PROBE_BW
```

Figure 17. BBR state transition log in bug-fixed 3.44 BBR program.

Besides, the timing of the state transition to ProbeRTT is different among flows. In flows 9 and 15, there is twenty second gap before the second or the third transition to ProbeRTT, respectively.

As depicted in Figure 11, the condition to execute `EnterProbeRTT()` includes `m_minRttExpired` is ture and `m_idleRestart` is false. We checked the value of `m_idleRestart` when becomes `m_minRttExpired` true. Figure 18 shows the result. As shown in this figure, `m_idleRestart` sometimes becomes true, and the moment when it is true corresponds to the moment when the BBR state transition skips the transition to RrobeRTT described in Figure 17. The condition that `m_idleRestart` is true is that there are no packets in flight when a TCP sender sends a packet during a congestion avoidance phase. That means the situation that `m_idleRestart` is true will be a normal situation even if the above mentioned bug is fixed.

```
flow 1 TcpBbr:CheckProbeRTT(): At 11.6166 m_idleRestart is false in BBR_PROBE_BW
flow 1 TcpBbr:CheckProbeRTT(): At 21.8414 m_idleRestart is false in BBR_PROBE_BW
flow 1 TcpBbr:CheckProbeRTT(): At 33.2152 m_idleRestart is true in BBR_PROBE_BW
flow 1 TcpBbr:CheckProbeRTT(): At 43.5469 m_idleRestart is true in BBR_PROBE_BW

flow 2 TcpBbr:CheckProbeRTT(): At 10.639 m_idleRestart is false in BBR_PROBE_BW
flow 2 TcpBbr:CheckProbeRTT(): At 20.9859 m_idleRestart is true in BBR_PROBE_BW
flow 2 TcpBbr:CheckProbeRTT(): At 31.9032 m_idleRestart is true in BBR_PROBE_BW
flow 2 TcpBbr:CheckProbeRTT(): At 42.3683 m_idleRestart is true in BBR_PROBE_BW

flow 3 TcpBbr:CheckProbeRTT(): At 10.8866 m_idleRestart is false in BBR_PROBE_BW
flow 3 TcpBbr:CheckProbeRTT(): At 22.0099 m_idleRestart is true in BBR_PROBE_BW
flow 3 TcpBbr:CheckProbeRTT(): At 32.0372 m_idleRestart is true in BBR_PROBE_BW
flow 3 TcpBbr:CheckProbeRTT(): At 42.3678 m_idleRestart is true in BBR_PROBE_BW

flow 4 TcpBbr:CheckProbeRTT(): At 10.8754 m_idleRestart is false in BBR_PROBE_BW
flow 4 TcpBbr:CheckProbeRTT(): At 21.092 m_idleRestart is false in BBR_PROBE_BW
flow 4 TcpBbr:CheckProbeRTT(): At 32.0601 m_idleRestart is false in BBR_PROBE_BW
flow 4 TcpBbr:CheckProbeRTT(): At 42.3687 m_idleRestart is true in BBR_PROBE_BW

flow 9 TcpBbr:CheckProbeRTT(): At 10.6391 m_idleRestart is false in BBR_PROBE_BW
flow 9 TcpBbr:CheckProbeRTT(): At 20.9822 m_idleRestart is true in BBR_PROBE_BW
flow 9 TcpBbr:CheckProbeRTT(): At 31.0323 m_idleRestart is false in BBR_PROBE_BW
flow 9 TcpBbr:CheckProbeRTT(): At 44.7737 m_idleRestart is true in BBR_PROBE_BW

flow 15 TcpBbr:CheckProbeRTT(): At 10.0241 m_idleRestart is false in BBR_PROBE_BW
flow 15 TcpBbr:CheckProbeRTT(): At 20.2643 m_idleRestart is false in BBR_PROBE_BW
flow 15 TcpBbr:CheckProbeRTT(): At 31.2293 m_idleRestart is true in BBR_PROBE_BW
flow 15 TcpBbr:CheckProbeRTT(): At 41.262 m_idleRestart is false in BBR_PROBE_BW

flow 16 TcpBbr:CheckProbeRTT(): At 10.7939 m_idleRestart is false in BBR_PROBE_BW
flow 16 TcpBbr:CheckProbeRTT(): At 21.0102 m_idleRestart is false in BBR_PROBE_BW
flow 16 TcpBbr:CheckProbeRTT(): At 32.0349 m_idleRestart is false in BBR_PROBE_BW
flow 16 TcpBbr:CheckProbeRTT(): At 43.6012 m_idleRestart is true in BBR_PROBE_BW
```

Figure 18. Log of `m_idleRestart` when `m_minRttExpired` is true in bug-fixed 3.44 BBR program.

D. *Results of Bug Fixing for Ns-3 Ver.3.40 BBR Program*

We applied the bug fixing to the ns-3 version 3.40 BBR program modified according to the Approach 2 described in the previous section.

Figure 19 shows the throughput of individual flows. By comparing with the result given in Figure 9, there is a big difference between the original program and the bug-fixed program. There is no unfairness among the throughput any more, but the throughput itself is extremely low.

Figure 20 shows the BBR state transition log for flows 1

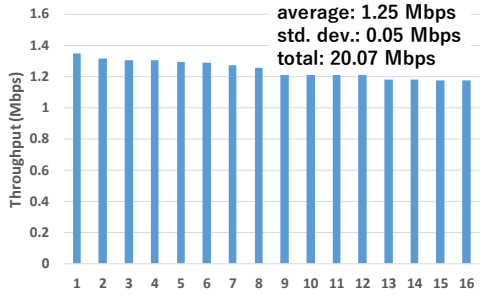


Figure 19. Throughput of individual flows using bug-fixed 3.44 BBR program ported into ns-3 version 3.40 (Approach 2).

```
flow 1 TcpBbr:SetBbrState(): At 0.024 Set to BBR_STARTUP
flow 1 TcpBbr:SetBbrState(): At 0.1682 Changing from BBR_STARTUP to BBR_DRAIN
flow 1 TcpBbr:SetBbrState(): At 0.1815 Changing from BBR_DRAIN to BBR_PROBE_BW
flow 1 TcpBbr:SetBbrState(): At 10.0272 Changing from BBR_PROBE_BW to BBR_PROBE_RTT
flow 1 TcpBbr:SetBbrState(): At 10.23 Changing from BBR_PROBE_RTT to BBR_PROBE_BW

flow 2 TcpBbr:SetBbrState(): At 0.024 Set to BBR_STARTUP
flow 2 TcpBbr:SetBbrState(): At 0.1682 Changing from BBR_STARTUP to BBR_DRAIN
flow 2 TcpBbr:SetBbrState(): At 0.1815 Changing from BBR_DRAIN to BBR_PROBE_BW
flow 2 TcpBbr:SetBbrState(): At 10.0273 Changing from BBR_PROBE_BW to BBR_PROBE_RTT
flow 2 TcpBbr:SetBbrState(): At 10.23 Changing from BBR_PROBE_RTT to BBR_PROBE_BW
...
```

Figure 20. BBR state transition log for 2 flows in Approach 2.

and 2. In both flows, the state transition to ProbeRTT is performed only once. After the state is back to ProbeBW, there are no state transitions any more. The other fourteen flows performed the similar state transitions.

In order to find the reason for this situation, we checked the value of `m_idleRestart` when becomes `m_minRttExpired` true, similarly with the case of ns-3 version 3.44. Figure 21 shows the log of the value change of `m_idleRestart`. At the timing around 10 sec., `m_idleRestart` is false and so the state can be shifted to ProbeRTT through function `EnterProbeRTT()` as shown in Figure 11. But, at the other timings, `m_idleRestart` is true and so the state transition does not occur. This is similar with the behaviors described in the previous subsection.

However, the situation in this case is a little different. In the case of the bug-fixed 3.44 BBR program, the log in Figure 18 shows that the event that `m_minRttExpired` becomes true happens regularly with a 10 sec. interval. But, in the case of porting bug-fixed 3.44 BBR program to ns-3 version 3.40, some events that `m_minRttExpired` becomes true are skipped. In flow 1, the ProbeRTT transition check around 45 sec. is skipped, and in flow 2, that around 31 sec. is skipped.

The condition that does not call function `EnterProbeRTT()` may be that `m_minRttExpired` is false, which is derived by the condition `tcb->m_lastRtt < m_minRtt`. Figure 22 shows the log for the cases when `tcb->m_lastRtt` is smaller than `m_minRtt`. In the case of this experiment, there are many cases that `tcb->m_lastRtt` is smaller than `m_minRtt`. In flow 1, there are fourteen such cases from time 20.5768 sec. to time 24.2652 sec. After that, the case that `tcb->m_lastRtt` is smaller than `m_minRtt` happens at time 34.9708 sec. The time gap between 24.2652 sec. and 34.9708 sec. is larger than 10 sec., and so the event that `m_minRttExpired` becomes true happens at time 34.6069 sec. (The state transition to ProbeRTT was not performed due to the values of `m_idleRestart`.) After time 34.9708 sec., several events that `tcb->m_lastRtt` is smaller than `m_minRtt` happened contiguously, and at each of them, the value of `m_minRttStamp` was updated. As a result, `m_minRttExpired` did not become true and the attempt to shift to ProbeRTT did not occur.

As for flow 2, after the `m_minRttExpired` event happens at time 25.2809 sec., the update of `m_minRttStamp` continued thirty-eight times from time 25.3807 sec. to time 35.3603 sec. So, the `m_minRttExpired` event around 25 sec. did not happen.

Those behaviors such that `m_idleRestart` is true and that `tcb->m_lastRtt` is smaller than `m_minRtt` can be feasible, and so the results in this experiment might be reasonable. The problem is that the bug-fixed 3.44 BBR program ported into ns-3 version 3.40 presents different behavior from that in bug-fixed ns-3 version 3.44.

```
flow 1 TcpBbr:CheckProbeRTT(): At 10.0272 m_idleRestart is false in  BBR_PROBE_BW
flow 1 TcpBbr:CheckProbeRTT(): At 20.322 m_idleRestart is true in  BBR_PROBE_BW
flow 1 TcpBbr:CheckProbeRTT(): At 34.6069 m_idleRestart is true in  BBR_PROBE_BW

flow 2 TcpBbr:CheckProbeRTT(): At 10.0273 m_idleRestart is false in  BBR_PROBE_BW
flow 2 TcpBbr:CheckProbeRTT(): At 25.2809 m_idleRestart is true in  BBR_PROBE_BW
flow 2 TcpBbr:CheckProbeRTT(): At 45.5526 m_idleRestart is true in  BBR_PROBE_BW
...
```

Figure 21. Log of `m_idleRestart` when `m_minRttExpired` is true in Approach 2.

```
flow 1 TcpBbr:UpdateRTprop(): At 20.5768 tcb->m_lastRtt is 0.0240235 m_minRtt is 0.0240269
... 12 events that tcb->m_lastRtt is smaller than m_minRtt
flow 1 TcpBbr:UpdateRTprop(): At 24.2652 tcb->m_lastRtt is 0.0240041 m_minRtt is 0.0240047
flow 1 TcpBbr:UpdateRTprop(): At 34.9708 tcb->m_lastRtt is 0.024117 m_minRtt is 0.0241337
flow 1 TcpBbr:UpdateRTprop(): At 35.3346 tcb->m_lastRtt is 0.0241024 m_minRtt is 0.024117
flow 1 TcpBbr:UpdateRTprop(): At 35.6985 tcb->m_lastRtt is 0.0240896 m_minRtt is 0.0241024
flow 1 TcpBbr:UpdateRTprop(): At 36.0624 tcb->m_lastRtt is 0.0240784 m_minRtt is 0.0240896
flow 1 TcpBbr:UpdateRTprop(): At 36.4542 tcb->m_lastRtt is 0.0240686 m_minRtt is 0.0240784
flow 1 TcpBbr:UpdateRTprop(): At 42.8648 tcb->m_lastRtt is 0.0240625 m_minRtt is 0.0240686
flow 1 TcpBbr:UpdateRTprop(): At 43.2849 tcb->m_lastRtt is 0.0240547 m_minRtt is 0.0240625
flow 1 TcpBbr:UpdateRTprop(): At 43.705 tcb->m_lastRtt is 0.0240479 m_minRtt is 0.0240547
flow 1 TcpBbr:UpdateRTprop(): At 44.1251 tcb->m_lastRtt is 0.0240419 m_minRtt is 0.0240479
flow 1 TcpBbr:UpdateRTprop(): At 44.5451 tcb->m_lastRtt is 0.0240367 m_minRtt is 0.0240419
flow 1 TcpBbr:UpdateRTprop(): At 44.9652 tcb->m_lastRtt is 0.0240321 m_minRtt is 0.0240367
flow 1 TcpBbr:UpdateRTprop(): At 45.3853 tcb->m_lastRtt is 0.0240281 m_minRtt is 0.0240321
flow 1 TcpBbr:UpdateRTprop(): At 45.8054 tcb->m_lastRtt is 0.0240246 m_minRtt is 0.0240281

flow 2 TcpBbr:UpdateRTprop(): At 10.0322 tcb->m_lastRtt is 0.024003 m_minRtt is 0.0240034
... 49 events that tcb->m_lastRtt is smaller than m_minRtt
flow 2 TcpBbr:UpdateRTprop(): At 10.912 tcb->m_lastRtt is 0.024 m_minRtt is 0.024
flow 2 TcpBbr:UpdateRTprop(): At 11.2827 tcb->m_lastRtt is 0.024034 m_minRtt is 0.0240388
flow 2 TcpBbr:UpdateRTprop(): At 11.4845 tcb->m_lastRtt is 0.0240297 m_minRtt is 0.024034
flow 2 TcpBbr:UpdateRTprop(): At 11.6864 tcb->m_lastRtt is 0.024026 m_minRtt is 0.0240297
flow 2 TcpBbr:UpdateRTprop(): At 11.8882 tcb->m_lastRtt is 0.0240228 m_minRtt is 0.024026
flow 2 TcpBbr:UpdateRTprop(): At 15.1525 tcb->m_lastRtt is 0.0240223 m_minRtt is 0.0240228
flow 2 TcpBbr:UpdateRTprop(): At 25.3807 tcb->m_lastRtt is 0.0240196 m_minRtt is 0.0240223
... 36 events hat tcb->m_lastRtt is smaller than m_minRtt
flow 2 TcpBbr:UpdateRTprop(): At 35.3603 tcb->m_lastRtt is 0.0240002 m_minRtt is 0.0240002
flow 2 TcpBbr:UpdateRTprop(): At 45.9444 tcb->m_lastRtt is 0.0240902 m_minRtt is 0.0241031
... 9 events hat tcb->m_lastRtt is smaller than m_minRtt
flow 2 TcpBbr:UpdateRTprop(): At 49.9188 tcb->m_lastRtt is 0.0240237 m_minRtt is 0.0240271
```

Figure 22. Log of cases when `tcb->m_lastRtt` is smaller than `m_minRtt` in Approach 2.

VI. CONCLUSIONS

This paper described some deficiencies of the TCP BBR software implemented in the ns-3 network simulator, which are detected when the transmission speed is high and heavy loads are applied to the bottleneck link. The experiment in this paper took a network configuration where sixteen senders and receivers are connected to the corresponding intermediate router through a 1 Gbps link and the intermediate routers are connected through a bottleneck 1 Gbps link. The buffer size in the bottleneck router is limited.

The first deficiency is that the TCP BBR software in different version ns-3 presents different behaviors. BBR in version 3.40 provided an unfairness in the throughput of individual flows. But BBR in version 3.44 provided a fair throughput.

We analyzed the BBR source code in 3.40 ns-3 and 3.44 ns-3, and found some differences. We modified the differences or ported 3.44 BBR code to 3.44 ns-3. But, the unfairness still remained. So far is the result of our previous conference paper [1].

Besides, we found that the BBR software in ns-3 does not perform the BBR state transition in a high speed environment [21]. In our experiment in this paper, this also happens. This is the second deficiency. In our network environment, ns-3 BBR software does not shift its state to ProbeRTT. We analyzed the ns-3 BBR source code and found a simple bug in the condition for entering ProbeRTT.

We fixed this bug and evaluated 3.44 ns-3 and 3.40 ns-3 to which 3.44 BBR program is ported. As for the bug-fixed

3.44 ns-3, the behaviors of sixteen BBR flows are reasonable and the state transitions to ProbeRTT are performed.

As for 3.40 ns-3 with 3.44 BBR ported, the situation was completely different. The unfairness of the throughput was resolved but the throughput itself became very low. The BBR state transitions were performed. The behavior itself might be reasonable, but the problem is that the behavior of 3.40 ns-3 with bug-fixed 3.44 BBR ported and that of bug-fixed 3.44 ns-3 are largely different. It may be said that the software in the ns-3 network simulator is not a real protocol program, and so the simulation results need to be examined carefully.

ACKNOWLEDGMENT

This work was supported by JSPS KAKENHI Grant Number JP25K15075.

REFERENCES

[1] T. Kato, T. Kato, R. Yamamoto, and S. Ohzahata, "Detailed Analysis of TCP BBR Implementation in Ns-3 Network Simulator under Heavy Traffic Loads,"The Seventeenth International Conference on Advances in Future Internet (AFIN 2025) IARIA, Oct. 2025, pp. 1-6.

[2] T. Sawada, R. Yamamoto, S. Ohzahata, and T. Kato, "Congestion Control Algorithm Estimation by Deep Recurrent Neural Network and Its Application to Web Servers on Internet," International Journal on Advances in Networks and Services, vol. 16, no. 1&2, pp. 27-35, 2023.

[3] S. Floyd, T. Henderson, A. Gurtov, and Y. Nishida, "The NewReno Modification to TCP's Fast Recovery Algorithm," IETF RFC 6582, 2012.

[4] S. Floyd, "HghSpeed TCP for Large Congestion Windows," IETF RFC 3649, 2003.

[5] S. Ha, L. Rhee, and L. Xu, "CUBIC: A New TCP-Friendly High-Speed TCP Variant," ACM SIGOPS Operating Systems Review, vol. 42, no. 5, pp. 64-74, 2008.

[6] D. Leith and R. Shorten, "H-TCP: TCP for high-speed and long distance networks," The Second International Workshop on Protocols for Fast Long-Distance Networks (PFLDnet 2004), Feb. 2004, pp. 1-16.

[7] L. Brakmo and L. Perterson, "TCP Vegas: End to End Congestion Avoidance on a Global Internet," IEEE J. Sel. Areas in Commun., vol. 13, no. 8, pp. 1465-1480, 1995.

[8] C. Fu and S. Liew, "TCP Veno: TCP Enhancement for Transmission Over Wireless Access Networks," IEEE J. Sel. Areas in Commun., vol. 21, no. 2, pp. 216-228, 2003.

[9] K. Tan, J. Song, Q. Zhang, and M. Sridharan, "A Compound TCP Approach for High-Speed and Long Distance Networks," The 25th IEEE International Conference on Computer Communicationsb (INFOCOM 2006), Apr. 2006, pp. 1-12.

[10] N. Cardwell, Y. Cheng, C. S. Gumm, S. H. Yeganeh, and V. Jacobson, V., "BBR: Congestion-Based Congestion Control," ACM Queue, vol. 14 no. 5, pp. 20-53, 2016.

[11] M. Hock, R. Bless, and M. Zitterbart, "Experimental Evaluation of BBR Congestion Control," The 25th IEEE International Conference on Network Protocols (ICNP 2017), Oct. 2017, pp. 1-10.

[12] K. Miyazawa, K. Sasaki, N. Oda, and S. Yamaguchi, "Cycle and Divergence of Performance on TCP BBR," The 7th IEEE International Conference on Cloud Networking (CloudNet 2018), Oct. 2018, pp. 1-6.

[13] D. Scholz et al., "Towards a Deeper Understanding of TCP BBR Congestion Control," IFIP Networking 2018, May 2018, pp. 1-9.

[14] S. Claypool, M. Claypool, J. Chung, and F. Li, "Sharing but not Caring – Performance of TCP BBR and TCP CUBIC at the Network Bottleneck," The Fifteenth Advanced International Conference on Telecommunications (AICT 2019) IARIA, Jul. 2019, pp. 74-81.

[15] H. Zhang et al., "Performance Analysis of BBR Congestion Control Protocol Based on NS3," The Seventh International Conference on Advanced Cloud and Big Data (CBD 2019), Sep. 201, pp. 363-368.

[16] G. Kim and Y. Cho, "Delay-Aware BBR Congestion Control Algorithm for RTT Fairness Improvement," IEEE Access, vol. 8, pp. 4099-4109, 2020.

[17] K. Lanigan, "An Evaluation of BBRv2 Congestion Control Using NS-3," Trinity College Dublin, 2020, availabe at https://publications.scss.tcd.ie/theses/diss/2020/TCD-SCSS-DISSERTATION-2020-046.pdf, accessed Jul. 2025.

[18] L. Tang, "BBR Fairness Evaluation Using NS-3," arXiv:2410.22560v1 [cs.NI], 2024, available at https://arxiv.org/html/2410.22560v1, accessed Jul. 2025.

[19] "Mininet An Instant Virtual Network on your Laptop (or other PC)," https://mininet.org/, accessed Jul. 2025.

[20] "ns-3 Network Simulator," https://www.nsnam.org/, accessed Jul. 2025.

[21] T. Kato, T. Kato, R. Yamamoto, and S. Ohzahata, "Does TCP BBR in Ns-3 Network Simulator Work Correctly ?" The 14th International Conference on Networks, Communication and Computing (ICNCC 2025), Dec. 2025, pp. 1-6.

[22] V. Jain, V. Mittal, ad M. P. Tahiliani, "Design and Implementation of TCP BBR in ns-3," Workshp on ns-3 (WNS3 2018), Jun. 2018, pp. 16-22.

[23] "ns-3 document, 32.3. Fifo queue disc," https://www.nsnam.org/docs/models/html/fifo.html, accessed Jul. 2025.

[24] "ns-3 document, 16.5.2.10. Loss Recovery Algorithm," https://www.nsnam.org/docs/models/html/tcp.html#loss-recovery-algorithms, accessed Jul. 2025.

[25] A. Aggarwal, S. Savage, and T. Anderson, "Understanding the Performance of TCP Pacing," The 19th IEEE International Conference on Computer Communicationsb (INFOCOM 2000), Mar. 2000, pp. 1157-1165.

[26] "ns-3: API and model change history," https://gitlab.com/nsnam/ns-3-dev/blob/ns-3.44/CHANGES.md, accessed Jul. 2025.

[27] A. Philip and N. Cardwell, "net-tcp_bbr: v3," https://github.com /google/bbr/blob/v3/net/ipv4/tcp_bbr.c, accessed Sep. 2025.