Lab3-Part2-Report

Qiushi Xu

qiushixu@usc.edu

In Part2 of lab3, I wrote code to incorporate the **BBR** Congestion Control Algorithm on the basis of cTCP implementation in Part1, which utilizes the **Bandwidth-Delay Product** as indicator of congestion condition, estimated by dynamic **min_RTT** and **max_BW** during the transmission.

1. Program Structure and Design

To implement BBR, the main work is inside ctcp_bbr.h and ctc_bbr.c files:

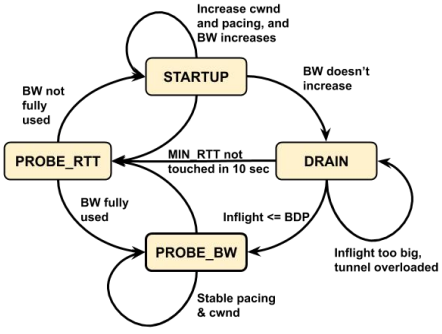
In ctcp_bbr.h, I defined the **bbr** structure which contains field needed for storing information of current BBR states, including:

a) **mode**: current mode among all four BBR states
b) **rt_prop**: estimated min_RTT in millisecond
c) **btlbw**: estimated max_BW, whose unit is bit/sec.
d) **num_rtt**: number of RTTs since the connection established
e) **min_rtt_stamp**: timestamp when min_RTT was last updated
f) **probe_rtt_done_stamp**: expected timestamp for the end of BBR_PROBE_RTT mode
g) **next_packet_send_time**: expected next packet send timestamp
h) **cycle_idx**: index of the current gain cycle of BBR_PROBE_BW mode
i) **cycle_start_time**: timestamp when the current PROBE mode cycle started
j) **cur_pacing_gain**: tracks the current pacing gain value
k) **cur_cwnd_gain**: tracks the current cwnd gain value
l) **full_bw_cnt**: number of rounds without BW growth, which may indicates full pipe

More Specifically, for btlbw, I introduced the **minmax_sample** structure from linux kernel, which I think is useful for storing and update maximum bandwidth samples.

Also I defined the **bbr_init()** function to initialize the bbr structure, and **check_bbr_state()** function to keep checking and updating the BBR state during transmission, for which I wrote implementation in ctcp_bbr.c.

The design logic for check_bbr_state() I wrote is the core component of BBR algorithm, which basically followed the instructions and operated the switching of states as the graph shown in the lab manual.



And let me explain what my codes did in each BBR phase:

a) **BBR_STARTUP**: set high pacing gain and cwnd gain to quickly fill pipe (like slow-start). If reached max_BW estimate plateaus, then switch to BBR_DRAIN mode

b) **BBR_DRAIN**: set low pacing gain to drain the queue created in STARTUP stage until the size of inflight data is smaller than estimated BDP, then switch into BBR_PROBE_BW mode

c) **BBR_PROBE_BW**: keep cycling pacing_gain to explore and fairly share bandwidth. Here I wrote a helper function **bbr_is_next_cycle_phase()** to judge whether it's appropriate to go to the next PROBE_BW cycle, based on the time difference between current time and recent cycle start time, as well as comparing inflight data volume with estimated BDP

d) **BBR_PROBE_RTT**: occasionally send slower to probe min RTT. I have to mention that All the other three states will fall into BBR_PROBE_RTT mode as long as min_RTT keeps untouched over 10 sec. And the BBR_PROBE_RTT mode will keep running for 200ms, then it will be switched back to either BBR_STARTUP or BBR_PROBE_BW depending on whether full BW reached

In addition, I made a little modification of cTCP on ctcp.c in order to enable BBR functionality.

In terms of data structures, I separated the unacked segments link list into new segments to be sent and retransmit segments, as re-sending segments are involved in our BBR process. Also I added some fields in ctcp_state which are useful for BBR, including inflight data size, app_limited_until(decides whether under app_limit region), last packet send timestamp and last packet sent size.

And I made primary work on the function **ctcp_send()** and **clean_acked_segments()**, which is called by ctcp_receive(). In ctcp_send(), besides re-sending old segments, for incoming segments, the send rate is controlled by conditions of inflight data upper limit and next packet send time. On the other hands, if there's no new segments, then set app_limit_until = inflight. After sending each segment, we increase the inflight, as well as set the next packet send time = now + segment size / (pacing gain * max_BW).

In clean_acked_segments(), I calculated the sample rtt based on the current time and the timestamp when segment was last sent, as well as the sample bandwidth with rtt and segment length. The min_RTT and max_BW were updated by sample rtt and sample bw. More specifically, if bw is not increasing, then do the increment of full_bw_cnt. Also, inflight and app_limit_until should be reduced. And check_bbr_state() is called to check and update the BBR state based on the latest sample statistics.

2. Implementation Challenges

One challenge during implementation is doing the modification of cTCP to incorporate BBR algorithm. At the first stage, I still utilized the unacked segments link list in the previous implementation, where I limit the sending rate with next packet send time. But later I realize that it may also block the re-sending segment when RTO expires. So I created a new link list to specially send and keep track of the re-sending segments.

Another challenge is implementing the logic of **check_bbr_state()**. I had to refer to the BBR kernel code patches and used some of the more subtle techniques such as how to decide whether the max_BW is reaching plateaus, as well as the tricks of trapping all the other states into BBR_PROBE_RTT mode.

3. Testing

First, I tested my BBR-CTCP on **dumbbell topology**, by running `lab3_topology.py` and script `bbr_mininet.sh`.

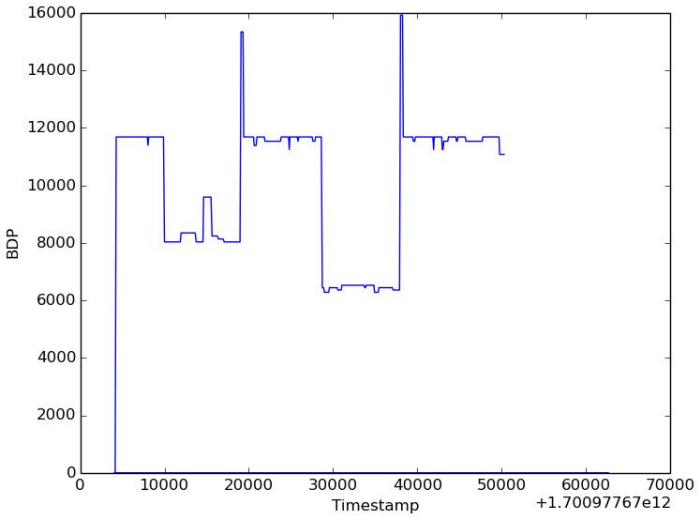
```
root@mininet-vm:~/cs551-651-labs-Apocalypse990923-qshi/lab3# sudo ./bbr_mininet.sh /home/cs551/cs551-651-labs-Apocalypse990923-qshi/
551 home folder: /home/cs551/cs551-651-labs-Apocalypse990923-qshi
Starting server1
Starting server2
Starting client1
Starting client2
Src File size: 810752 Bytes
Ending all nodes
Dst1 File size: 810753 Bytes
Dst2 File size: 810753 Bytes
SEND_TIME1: 24 sec
SEND_TIME2: 24 sec
Dumbbell Throughput: 270250 bps
```

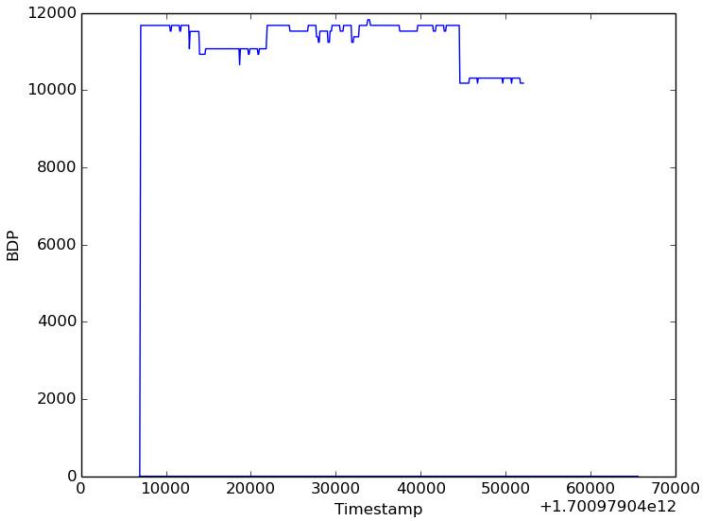
Note that he output might be slight different because I did a little modification on bbr_mininet.sh in order to print the more information for debugging, such as file size and send time.

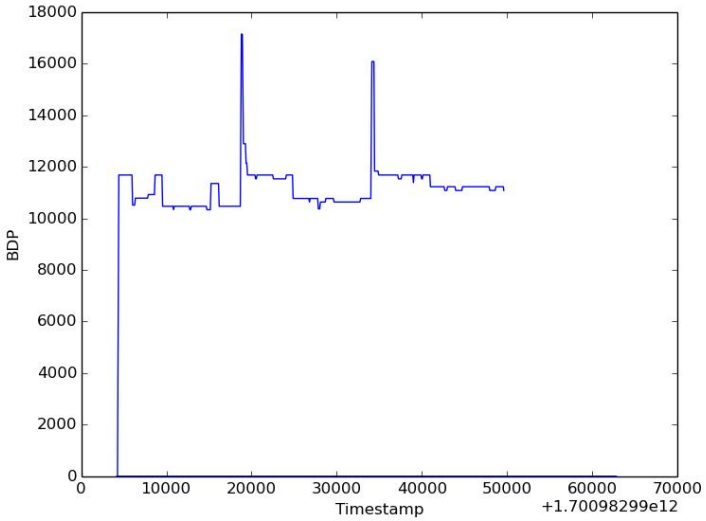
In addition, I also did the test on I2 topology, by putting ctcp in lab2 folder and running `bbr_i2.sh`. As there's problem with the original I2 script, so I just commented out server2 and client2 and calculate throughput by performance of transmission between server1 and client1.

```
START SEND TIME: 1701030561
Src File size: 810752 Bytes
Ending all nodes
ovs-controller: no process found
ctcp: no process found
SEND END TIME1: 1701030608
Dst File size: 810753 Bytes
SEND_TIME: 47 sec
I2 Throughput: 138000 bps
```

These are the three results of BDP plot executed by bbr_i2.sh.







In addition, I run the ctcp_test of Lab3-PartA, which even though is not required to be tested in partB. I found that it will always fail in Handles sliding window and Supports different send/receive windows tests. The reasonable explanation I can give is that sliding window test wants to fulfill the whole window size, while the BBR algorithm controls the sending rate, thus preventing the inflight data from bursting the sending window.

4. Remaining Limitations

1) In PROBE_RTT phase, even though I send segments in lower rate to reduce segments inflight, I still cannot ensure that there are only 4 segments inflight.

2) Also, some complex techniques are absent in my BBR algorithms, compared with patch codes, such as token-bucket policer mitigation logic.

3) In addition, for BDP plot graph, it seems not a perfect illustration of a BBR process, as you can see that the BDP is a sudden surge in the startup phase, not a gradual increase.