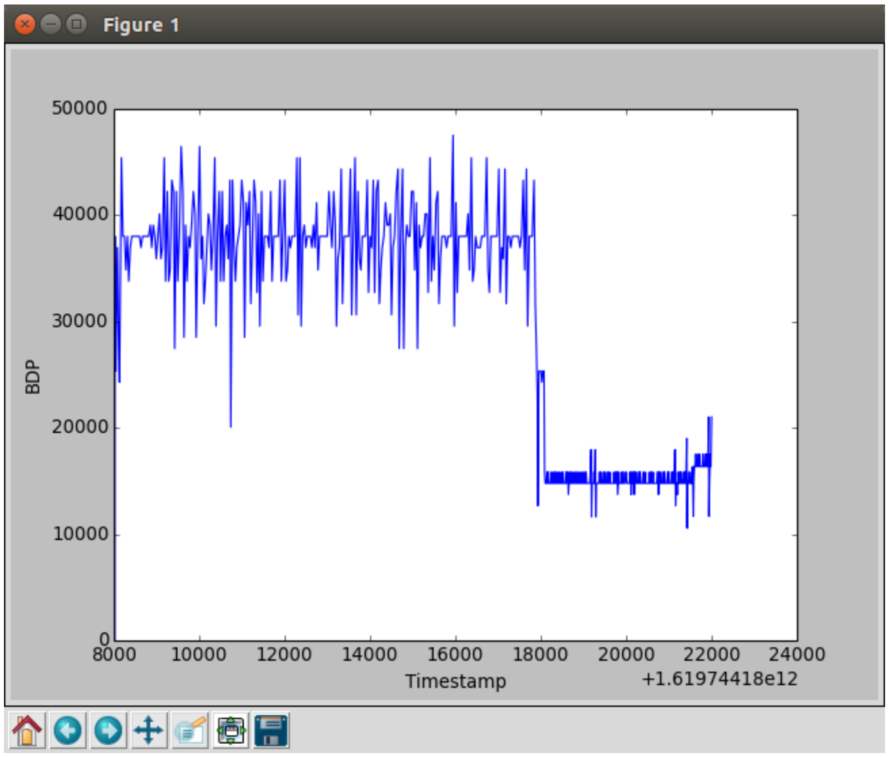
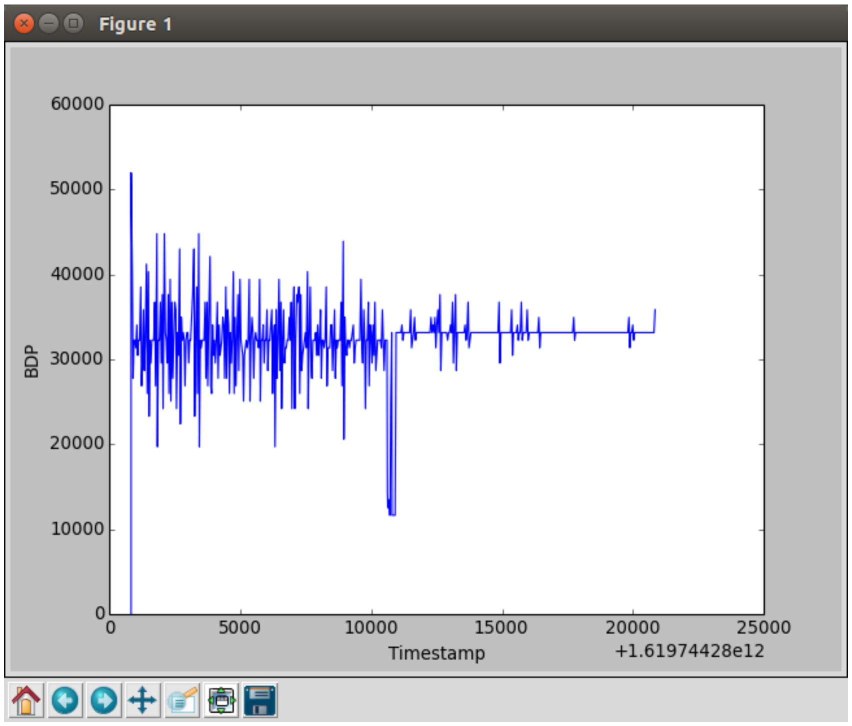
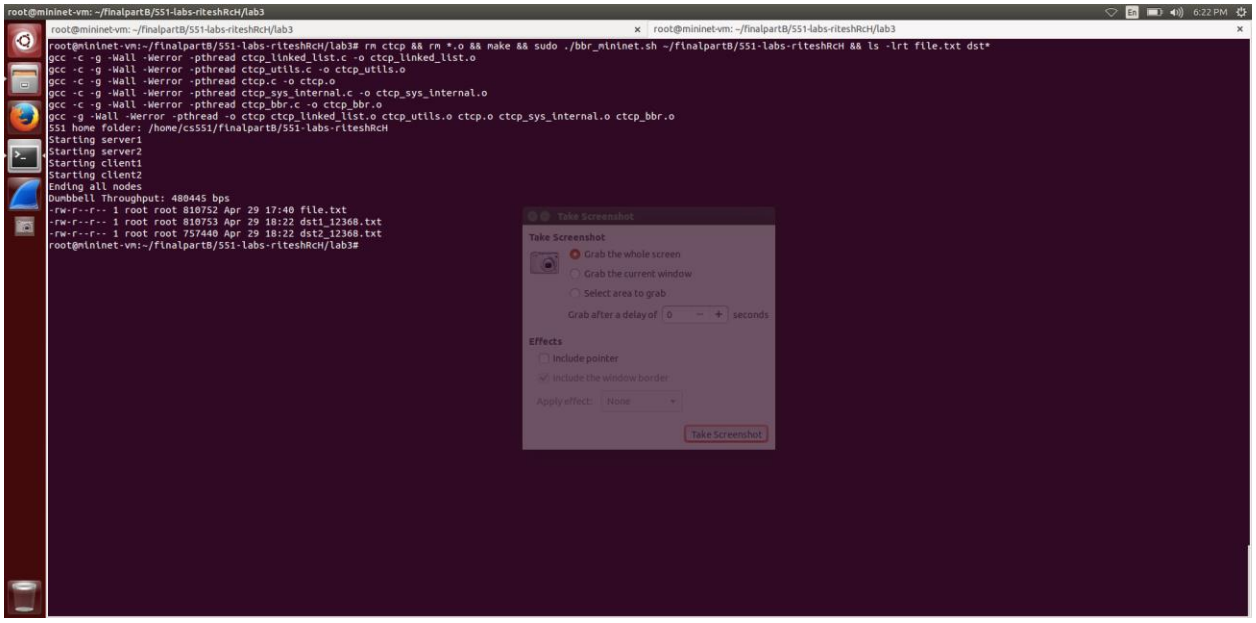
Lab 3 Part 2 Report

In this lab, we had to build our own transport layer protocol with a customized algorithm. Initially in the previous part we built regular TCP state machine logic for the transport protocol, however in this lab we modified the TCP algorithm with BBR logic from BBR algorithm by Google. Most popular variants of TCP such as TCP Tahoe, TCP Reno, TCP Vegas etc. use either packet loss/timeout or 3 duplicate ACKs as the mechanism to detect congestion in the network. These indicators are not true indicators of congestion, they can be thought of as symptoms instead of indicators of congestion. BBR algorithm is an algorithm developed and deployed by Google which stands for 2 parameters upon which the algorithm is built around namely: Bottleneck Bandwidth and Round trip propagation. It is basically trying to achieve the optimal point of operation by adjusting the sender's rate using parameter called pacing gain in order to fill the whole "network pipe" which we term is as "Bandwidth Delay Product" or BDP for short.

Following this concept, we keep track of maximum bandwidth we can capture along with the minimum RTT we can capture. Both of these help us to fill the network pipe without overflowing intermediate router buffers and also not overwhelm or starve the receiver at the same time. We have 4 states in BBR namely: STARTUP (this is similar to slow start where we exponentially fill the pipe and we stop when bandwidth remains still), DRAIN (we drain the queue we created in STARTUP state), PROBE_BW (try out pacing gain values to explore the bandwidth available and estimate bottleneck bandwidth), PROBE_RTT (phase to get minimum RTT). In terms of code structure, we mainly have 4 files: ctcp.c/h and ctcp_bbr.h/c where header files have declarations and c files have implementations and in ctcp_bbr.c the code keep tracks of various parameters required in the BBR algorithm. Ctcp.c has TCP code into which BBR has been integrated and BBR algorithm (specifically state change) is invoked when an ACK is received in the TCP code. Major implementation challenges were making sure the existing TCP code did not break when integrating BBR code along with calculation and transitioning of states in the BBR algorithm. Below are screenshots of various tests that we had to perform in order to test our BBR algorithm:







root@mininet-vm: ~/finalpartB/551-labs-riteshRcH/lab3
root@mininet-vm: ~/finalpartB/551-labs-riteshRcH/lab3
root@mininet-vm: ~/finalpartB/551-labs-riteshRcH/lab3
6:22 PM
Take Screenshot
Take Screenshot
Grab the current window
Select area to grab
Effects
Include pointer
Apply effect:
None
Take Screenshot

```
root@mininet-vm:~/finalpartB/551-labs-riteshRcH/lab3# rm ctcp && rm *.o && make && sudo ./bbr_mininet.sh ~/finalpartB/551-labs-riteshRcH && ls -lrt file.txt dst*
gcc -c -g -Wall -Werror -pthread ctcp_linked_list.c -o ctcp_linked_list.o
gcc -c -g -Wall -Werror -pthread ctcp_utils.c -o ctcp_utils.o
gcc -c -g -Wall -Werror -pthread ctcp.c -o ctcp.o
gcc -c -g -Wall -Werror -pthread ctcp_sys_internal.c -o ctcp_sys_internal.o
gcc -c -g -Wall -Werror -pthread ctcp_bbr.c -o ctcp_bbr.o
gcc -g -Wall -Werror -pthread -o ctcp ctcp_linked_list.o ctcp_utils.o ctcp.o ctcp_sys_internal.o ctcp_bbr.o
551 home folder: /home/cs551/finalpartB/551-labs-riteshRcH
Starting server1
Starting server2
Starting client1
Starting client2
Ending all nodes
Dumbbell Throughput: 480445 bps
-rw-r--r-- 1 root root 810752 Apr 29 17:40 file.txt
-rw-r--r-- 1 root root 810753 Apr 29 18:22 dst1_12368.txt
-rw-r--r-- 1 root root 757440 Apr 29 18:22 dst2_12368.txt
root@mininet-vm:~/finalpartB/551-labs-riteshRcH/lab3#
```

[✓ Grab the whole screen]

Grab after a delay of 0 seconds

[✓ Include the window border]