An Pham
9477698118

CSCI 551 Lab 3 Part 2 Report

Overview: For lab3b I implemented Google's BBR as congestion control for the ctcp project. Instead of using the traditional packet loss based methodology, my implementation of congestion control measures the maximum bottleneck bandwidth and minimum round-trip propagation delay to calculate bandwidth delay product (BDP). I followed Google's code patch for the Linux kernel; however, my implementation had to interoperate with the ctcp project, which is a user level transport layer, supported by the custom CSCI 551 vm image.

Tools & Environment:

- Virtual Box: Custom CSCI 551 image, which supports tcp changes at the user level
    - We use this environment to test our code
- Visual Studio Code:
    - Text editor with various functionalities, providing ease of development
    - Ssh-remote: to avoid using VM's GUI
- GDB:
    - Allows us to debug our code
- Valgrind:
    - Allows us to detect memory leaks within our code

Program Structure and Design:

- **bbr_ctcp.h/.c:** contains all the function prototype, macros, constants, structure, enumerations, and function implementations related to BBR
- **ctcp.c:** Does #include bbr_ctcp.h to utilize the data structures, functions, and constants related to bbr.
- **ctcp_sys_internal.c:** I modified the tcp_handshake function to keep looping for a SYN-ACK

Design & Implementation:

1. Obtain high throughput:
    - Similar to a control system, our throughput can overshoot or undershoot, thus tweaking pacing gains & congestion window gains can help us balance around the optimal throughput
2. State Machine Pattern:
    - In a nutshell, our bbr logic follows closely to the state machine shown in the lab manual.

3. Modular integration:
    - We want to decouple our bbr code from the ctcp code as much as possible. This will allow us to track down if bugs are related to our bbr code or the ctcp implementation.

Function Implementations:

- **bbr_init(bbr):** similar to ctcp_init(), we allocate memory for our bbr structure, initialize the structure's values, and return a pointer of the allocated structure to the caller.
- **bbr_update_model(bbr):** This function forwards the bbr structure to the appropriate function handler (based on current bbr_mode)
- **bbr_startup_state(bbr):** This function checks if there has been at least a 25 percent growth in bandwidth, during the startup phase. If bandwidth fails to meet a significant growth, we change the bbr_mode to BBR_DRAIN.
- **bbr_drain_state(bbr):** This function drains 4 packet acknowledgement rounds, since startup_state probes for 3 packets past the optimal point. This will put us near the optimal point.
- **bbr_probe_bw_state(bbr):** This function loops around our array of pacing values & updates the current pacing gain based on the bdp.
- **bbr_probe_rtt_state(bbr):** This function restricts our pacing gain as well as our congestion window to update rtt value.
- **bbr_update_bw(bbr, round_trip_time, seg_len):** This function calculates & returns the bandwidth given the round_trip_time and segment_length of a packet
- **bbr_update_rtt(bbr, round_trip_time):** This function updated the round trip time if we've found a shorter round_trip_time or changes the current mode to BBR_PROBE_RTT if it's time to change modes.

Updates to ctcp:

- **ctcp_segment_with_info_t:** we added is_in_flight & is_app_limited flags into this structure to help us determine which packets are app_limited & packets are currently in flight.
- **ctcp_state_t:** we added a pointer to a bbr structure, FILE pointer to record the bandwidth delay product measurements, and timestamps to track when the previous packet was sent & when ctcp_timer goes off.
- **ctcp_receive():** Whenever we received an ACK packet, we needed to call update_bbr_model(). In addition, we calculate and record round trip time & bandwidth of the acknowledged packet.
- **ctcp_timer():** Because this function is called periodically, we updated the function to check if 10 seconds elapsed. If roundtrip propagation delay hasn't been updated for 10 seconds, then we need to change the mode to BBR_PROBE_RTT.

- **ctcp_send_sliding_window():** We updated this function to check if the packet is app_limited and if it's time to send the packet. After sending, we update the packets accordingly.

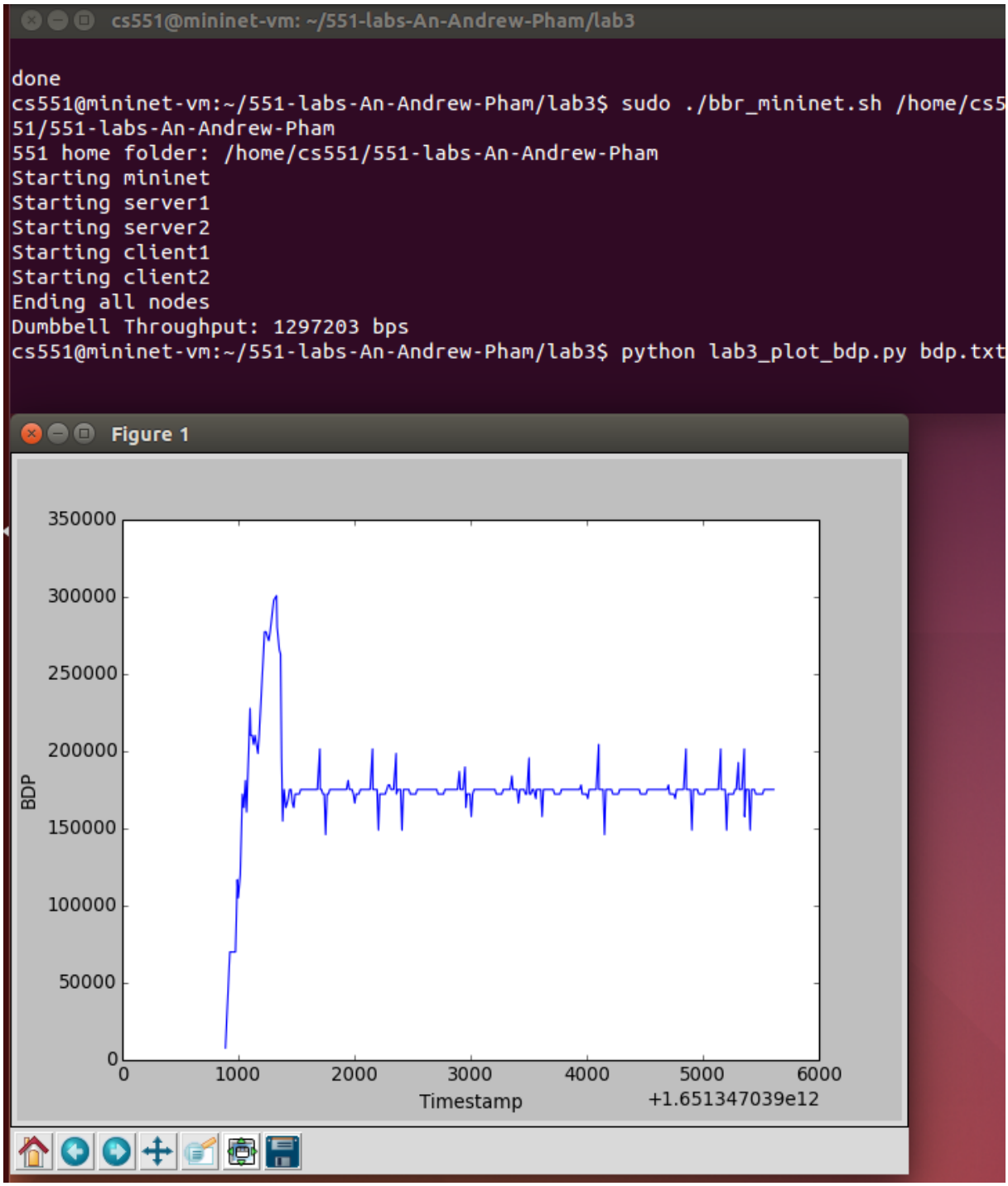
Implementation Challenges: One of the challenges of this lab is the environment dependencies. Sometimes we have to delete and create a fresh image of the 551 vm, which could be time consuming. Another challenge is bugs from lab3a propagating to our lab3b project. Without solving bugs from our previous lab, it would be difficult to determine if the bug is from our bbr implementation or our ctcp flow control implementation.

Testing:

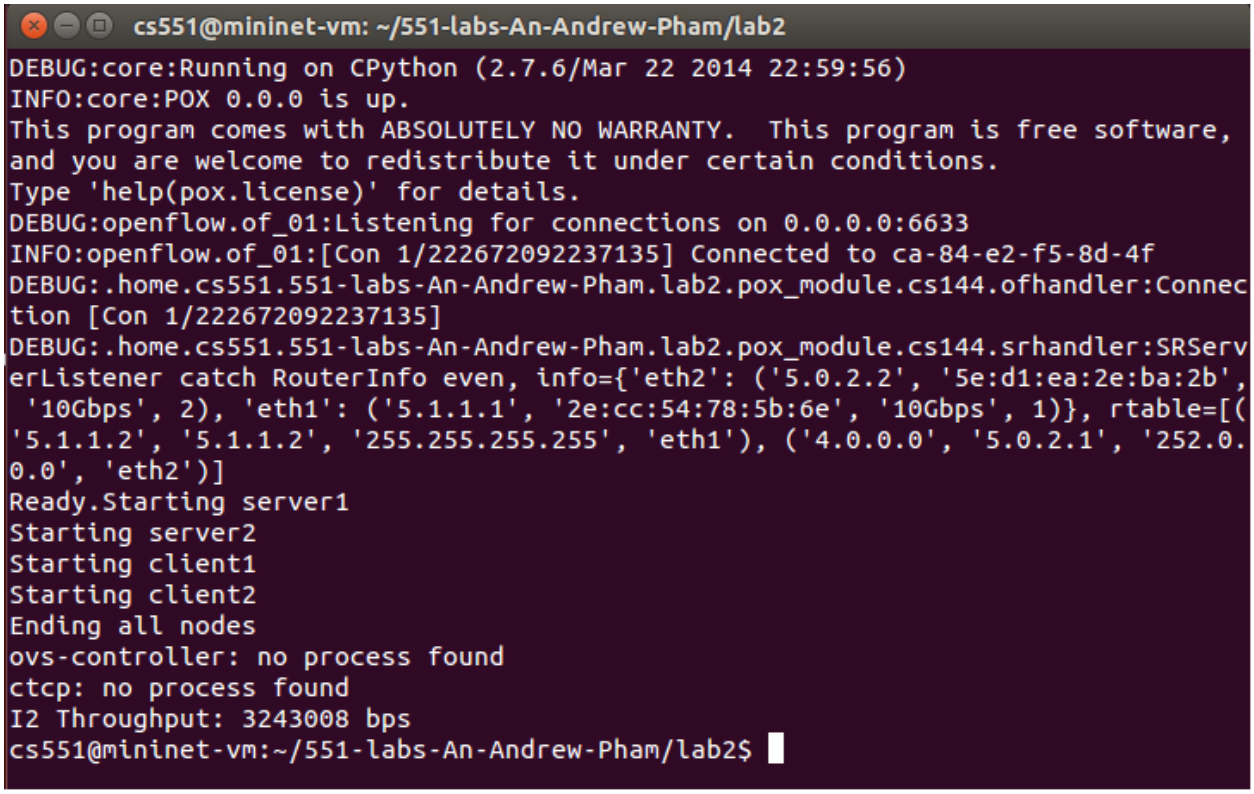
- **Dumbbell Topology:** I tested the dumbbell topology with the "bbr_mininet.sh" script provided by the slides, instead of using the default file provided by the lab3 folder. I was able to obtain a throughput of 1.2Mbps; however, this number changes between runs.
- **I2 Topology:** I tested the i2 topology with the default "bbr_is.sh" script and obtained a throughput of around 3.2Mbps. Similar to the dumbbell topology, this number changes between runs. The i2 topology test introduced more variables that could skew with our results such as a correct sr implementation or having the routers configured correctly.
- **Manual testing on mini-net:** Use lab3_topology.py to create the mininet topology, then start client & server by hand. This allows us to use gdb for debugging and check for memory leaks with valgrind.
- **fprintf(stderr):** Changing line 350 & 363 within run_test.py to create a /tmp/client_logs & /tmp/server_logs provided us with extra information with where our bug might be within the code. This allowed us to see fprintf(stderr) statements.

Throughput Results & Plots:

Dumbbell Topology:



I2 Throughput:



Remaining Bugs:

- The throughput value is strangely high. Files sometimes come as incomplete for the i2 topology.
- The throughput value can be inconsistent & would require multiple runs. Future work requires more stability. The throughput value should be close to each other.