Performance Evaluation Assignment

Chirnogeanu Maria-Andreea - 342C5

1 Requirements

1.1 Set up the network simulation

After running the experiments, I created three plots: the first one shows the tcp bitrate for destination (h3), the second one compares the tcp and udp bitrate results for destination (h3) and the third one aims to compare the first experiment with the second one.

The plots are shown below:

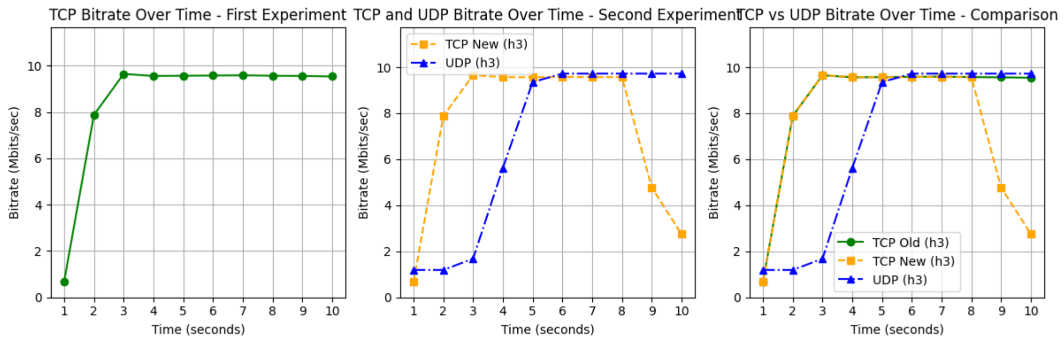


Figure 1: Experiment plot results

In the first experiment, where only TCP traffic was present, the connection showed stable and consistent performance. The bitrate started low in the first second but quickly ramped up to approximately 9.6 Mbit/s by the third second and remained steady for the rest of the 10-second interval. This indicates a well-functioning TCP connection with effective congestion control and no significant packet loss.

In the second experiment, both TCP and UDP traffic were transmitted simultaneously. While UDP maintained a stable bitrate close to 9.6 Mbit/s after the initial ramp-up, TCP performance declined in the later seconds of the test. Although TCP initially reached the same throughput level as in the first experiment, its bitrate began to drop after the seventh second and fell to around 2.8 Mbit/s by the end of the test. This degradation is likely due to competition for bandwidth, where the UDP stream—unregulated by congestion control—overwhelmed TCP, which reduced its sending rate in response to packet loss or increased congestion.

Overall, these results highlight how the presence of aggressive UDP traffic can negatively impact TCP performance in a shared network environment.

1.2 Implement connection monitoring tool

In this exercise, I made a series of modifications to the `socket_diag.c` code, which handles data transmission via sockets.

TODO 1: To extract `tcp_info`, I modified the `idiag_ext` parameter as described here.

TODO 2: I set the source IP (in our case, `h1`), port, and destination IP (in our case, `h3`). Why not the source port? The source port changes with every run of the `iperf3` command, so it doesn't need to be set manually in the code.

TODO 3: Initially, I modified the code to display the `tcp_info` fields I was interested in:

- `tcpi_rtt` – the average round-trip time.
- `tcpi_rttvar` – the RTT variation, indicating connection stability.
- `tcpi_snd_cwnd` – the maximum number of segments that can be sent without acknowledgment.
- `tcpi_snd_wnd` – the send window imposed by the receiver, a flow control mechanism.
- `tcpi_bytes_sent` – all bytes sent, including retransmissions.
- `tcpi_delivery_rate` – the actual data delivery rate in bytes per second.

After verifying that monitoring of the selected metrics worked correctly, I extended the logic to observe activity over a full 60 seconds. I added a `for` loop and a one-second `sleep` at the end of each iteration (without this sleep, the data flow is not monitored properly).

I created a CSV file (named `tcp_stats`) to store the monitored data, which I later used to generate plots. I then modified the code to accept the CSV filename as an argument, which helps when automating the data extraction process.

The `socket_diag.c` script monitors activity over the full 60-second duration. This includes sending and responding to messages, extracting the payload, and displaying the relevant metrics mentioned above. To place the `for` loop appropriately, I had to reorganize some of the code (moving all parts related to `conn_req`, including TODO 1 and TODO 2) and change the error handling logic within the loop. Instead of using `goto`, I refactored the code to use `break;` and removed the `out` label, setting `res = 0` at the end of the `main` function.

After running the code, I noticed that some columns were repeated. This was caused by insufficient filtering (I had not included the source port). Initially, I removed these duplicates manually, but later automated the process. Finally, I wrote a script to generate plots for each metric based on the second at which it was recorded.

I got the following results:

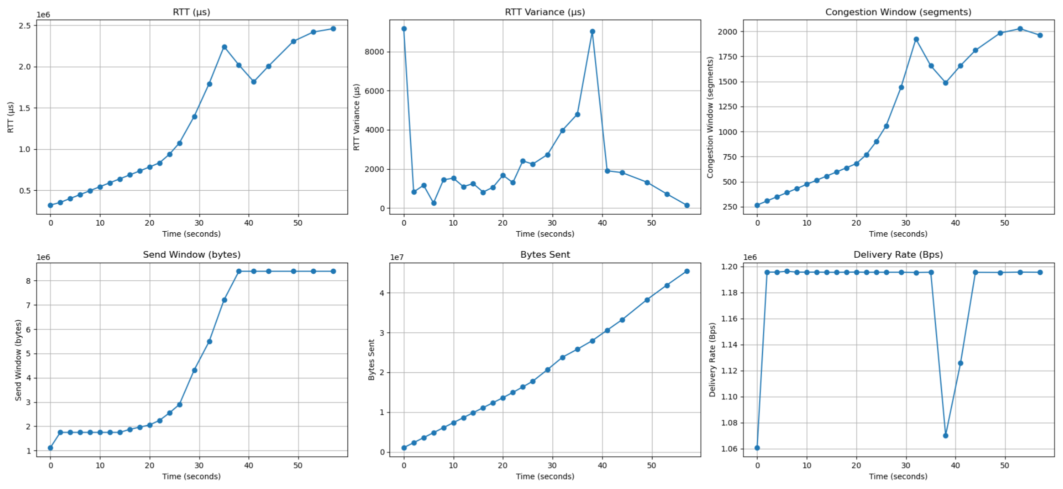
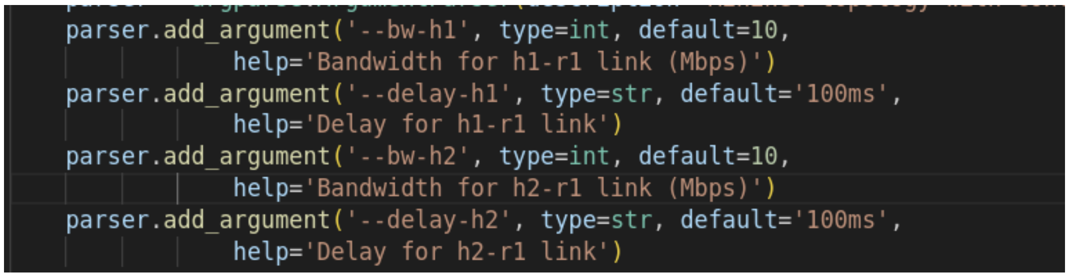


Figure 2: Plots generated as a result of running the code

1.3 Differential analysis

Here, I noticed that there are arguments in the `topology.py` code that can be modified when running the script, specifically in the following section:



```
    parser.add_argument('--bw-h1', type=int, default=10,
                        help='Bandwidth for h1-r1 link (Mbps)')
    parser.add_argument('--delay-h1', type=str, default='100ms',
                        help='Delay for h1-r1 link')
    parser.add_argument('--bw-h2', type=int, default=10,
                        help='Bandwidth for h2-r1 link (Mbps)')
    parser.add_argument('--delay-h2', type=str, default='100ms',
                        help='Delay for h2-r1 link')
```

Figure 3: Arguments of the `topology.py` script

For this analysis, I decided to assign different values to each parameter of the `topology.py` command shown above. I assigned three different values to each argument and generated CSV files in folders named after the modified parameter, located inside the `results` directory. The results obtained are later compared to the baseline results stored in `csv_stats`.

I created two scripts: one to automatically generate the CSV files, named `run_experiments.sh` (which runs the `iperf3` commands inside Mininet, compiles, and runs the code), and another to clean the files of duplicate entries, named `clear_csv.sh`.

Finally, I created plots for all the metrics, where I analyze bandwidth and delay, comparing the results with the results from `csv_stats`.

The results of this experiment are presented below:

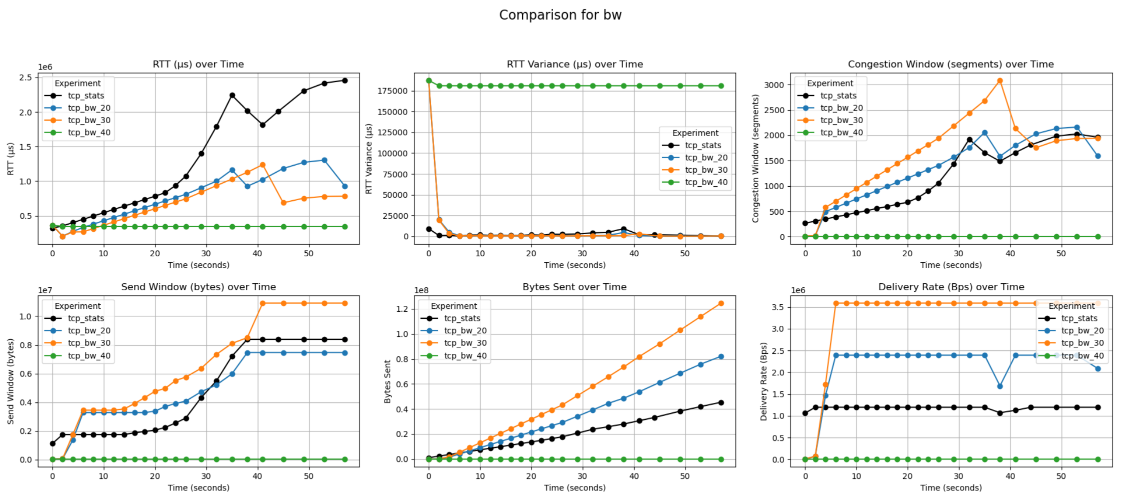


Figure 4: Results of modifying the bandwidths for h1-r1 and h2-r1

RTT (µs) over Time

- Higher bandwidth reduces queuing delay, resulting in lower RTT. In contrast, lower-bandwidth links experienced queue buildup, leading to increased RTT. The 40 Mbps configuration maintained the lowest and most stable RTT, suggesting minimal congestion.

RTT Variance (µs) over Time

- The elevated RTT variance observed in the 40 Mbps configuration may indicate measurement instability or jitter due to the absence of regular data flow. Other configurations remained more consistent.

Congestion Window (segments) over Time

- The 30 Mbps configuration enabled the TCP sender to ramp up its sending rate most effectively. The flat congestion window in the 40 Mbps case points to a misconfiguration or a failure in traffic generation.

Send Window (bytes) over Time

- The 30 Mbps configuration allowed the largest and most stable send window, supporting better throughput. The 40 Mbps configuration remained at zero, reinforcing the likelihood of a setup error.

Bytes Sent over Time

- Higher bandwidth increased throughput up to 30 Mbps. The complete absence of data sent in the 40 Mbps test confirms that no transmission occurred, likely due to incorrect configuration or command failure.

Delivery Rate (bps) over Time

- Delivery rate reflects effective throughput. The 30 Mbps setup yielded the best delivery performance, while the 40 Mbps configuration failed to deliver any data, further supporting the need for troubleshooting.

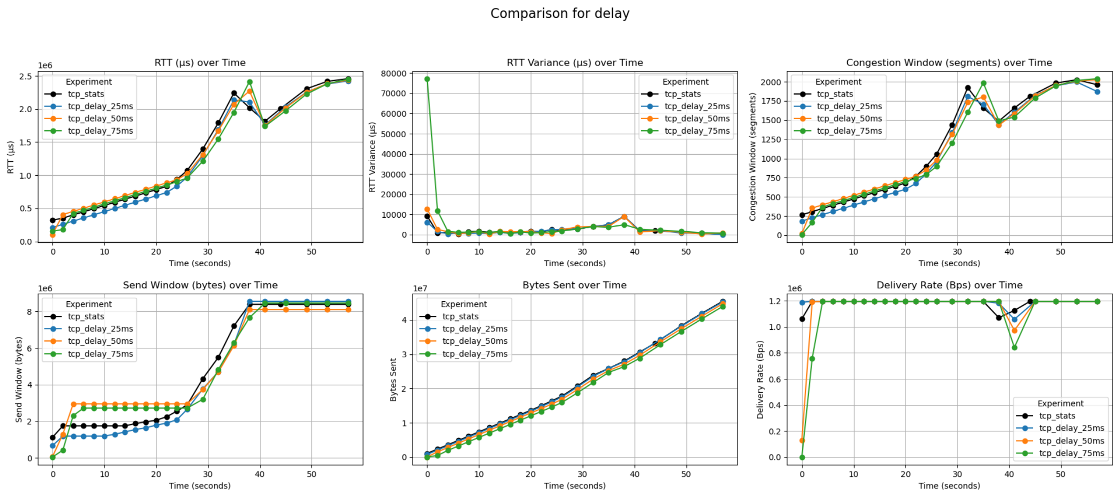


Figure 5: Results of modifying the delays for h1-r1 and h2-r1

RTT (µs) over Time

- Added delay directly increases end-to-end RTT, as expected. The baseline recorded the lowest RTT, while `tcp_delay_75ms` showed the highest, reflecting the linear impact of artificial delay on round-trip measurements.

RTT Variance (µs) over Time

- RTT variance remained low and stable after initial convergence in all cases, with a temporary spike in the 75 ms configuration. This may indicate transient queue formation during TCP slow start, but it had minimal long-term impact.

Congestion Window (segments) over Time

- Despite increasing delay, all configurations achieved comparable congestion window growth. This shows that TCP congestion control adapts effectively even under higher latency, maintaining performance consistency by the end of the session.

Send Window (bytes) over Time

- Higher delay delayed the growth of the send window initially, but all flows converged to similar window sizes. This indicates that TCP was ultimately able to utilize available capacity, though with slower ramp-up under increased latency.

Bytes Sent over Time

- While slightly affected by delay, the total number of bytes sent remained consistent across experiments. Higher delays led to marginally reduced throughput, but did not significantly restrict the volume of data transferred over the full duration.

Delivery Rate (bps) over Time

- All configurations reached stable delivery rates shortly after flow initiation. Delay had limited long-term effect on delivery rate, though short dips—possibly due to retransmissions—were visible across all experiments.

1.4 Evaluate bpftune impact

I installed `bpftune`, then ran the command `sudo ./bpftune tune tcp-buffer` and repeated the second experiment from Task 2.1.

The results of the experiment can be seen below:

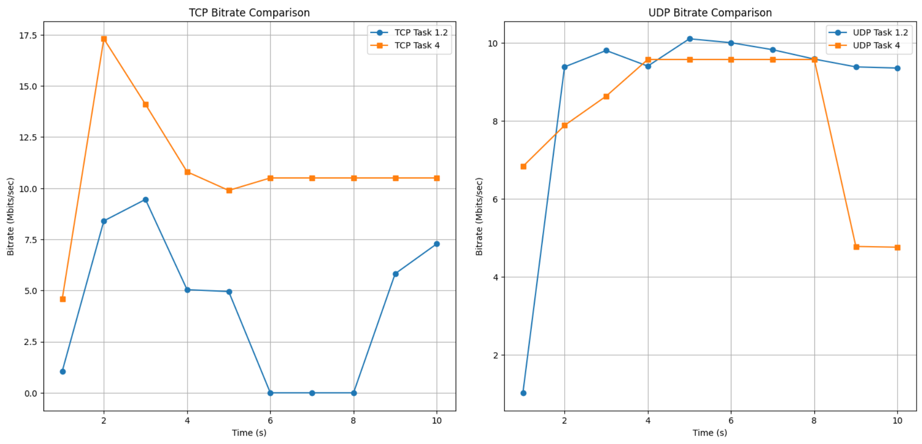


Figure 6: Results of the experiment with bpftune

I noticed that using `bpftune` had a noticeable but negative effect: both TCP and UDP performance were significantly impacted, indicating that the applied tuning profile was not suitable for the test scenario. The high losses and reduced bitrate suggest that the buffers were suboptimally configured. This highlights the importance of context-aware tuning strategies.

To understand why this happened, I examined the code in the file `tcp_buffer_tuner.bpf.c`. I noticed that fixed or overly restrictive TCP buffer values were being applied, without accounting for the network context or the presence of UDP traffic. This tuning strategy leads to artificial congestion, low bitrate, and significant retransmissions or packet loss.