CMPUT 313 – Computer Networks **Winter 2025**

Assignment 2 – Sizing router buffers in the Internet

This assignment is due **March 11, 2025, at 5:00pm** Mountain Time.

Objective

In this assignment you will investigate how buffer sizing affects network performance and understand the sawtooth behaviour of TCP congestion control. You will also learn about networking tools, such as iperf, ping, curl, and tc.

Overview

As discussed in class, TCP provides end-to-end reliability over an unreliable network. This means that a sender that uses TCP must ensure that the packets it sends are received correctly by the receiver. TCP provides end-to-end reliability using sequence and acknowledgment numbers, a retransmission timer, and windows for pipelining.

The size of the sender window determines the maximum number of packets that can be sent without waiting for an acknowledgment for each one. Thus, it controls the transmission rate of the sender. If the window size is too large, the sender will cause buffer overflow at routers located on the path to the receiver, because they are not able to forward packets as fast as they arrive. To avoid this, TCP adapts the congestion window size (`cwnd`) to the network congestion state, where the congestion is detected using either round-trip time (`RTT`) or packet loss rate. There are many algorithms for controlling the value of `cwnd`, all with the goal of maximizing the connection's throughput while avoiding congestion in the network.

Increasing the amount of buffer space available to routers might seem to be a good strategy to address the overflow problem. Yet, excess buffering, known as bufferbloat, largely defeats TCP's congestion control mechanisms, which rely on timely packet drops or reliable `RTT` estimates to detect congestion. In this assignment, we will investigate the effect of different buffer sizes on packet delay and throughput/goodput of the network. We consider a simple network that consists of two hosts and a router in between.

In a real network, measuring the host congestion window and the router buffer occupancy, and aggregating this information in one place is difficult as they are private to the sender and the router, respectively. Further, it would be very difficult to vary things like buffer sizes and link capacities with real hardware. Emulating a network is a much better approach for getting reproducable metrics easily. We are going to emulate a small network (depicted in Figure 1) in Mininet. The virtual devices in the emulated network, such as hosts and switches, can run any program or utility that can run on Linux!

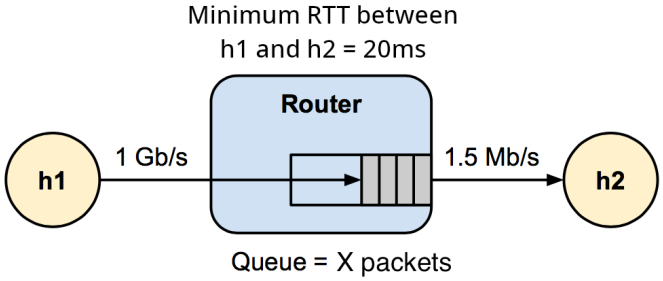

Figure 1: Small model network. h1 is a Web server with a fast link to the Internet. h2 is a client at home.

Mininet is already installed in your VM. A Mininet tutorial is available on their official website, though it focuses on the command line API. We will instead use the much more powerful Python API, which is documented here.

However, the documentation is quite minimal, so you should go over the worksheet for Lab 5 to learn its Python API before starting to work on the assignment.

Starter Code

The code on GitHub classroom provides the following files:

- `main.py`: The bufferbloat experiment code. This will setup a network, establish connections between client and server, and record performance metrics once it is complete.
- `plotting.py`: Converts the raw data captured by `main.py` into a chart.
- `run.sh`: Runs the bufferbloat experiment code with various congestion algorithms and buffer sizes. This may take more than a couple of minutes.
- `lossy_run.sh`: Identical to `run.sh`, but names the output directories with `lossy_` prefix. It will be used in Part 3.
- `webserver.py` and `index.html` are both required for `main.py`. The Web server responds to HTTP GET requests for the index page.

Note that the provided code is not complete. You will need to make several modifications to `main.py` before things will run.

To run the experiments, simply run the bash script: `bash run.sh`. This will run experiments with two different buffer sizes (10 and 50 packets) and 3 different TCP congestion control algorithms, namely TCP Reno, CUBIC TCP, and TCP BBR. The results will be stored in a directory called `data_<algo>_<buffer_size>`, with those variables replaced.

The code will setup a network using Mininet, continually send a high volume of TCP traffic using iperf, send a ping at regular intervals to measure `RTT`, and periodically make webpage requests using HTTP over TCP to measure how long it takes to fetch it. Since the network becomes congested by iperf, we expect the router buffer to fill up during these bursts of requests. The code records four important statistics, which are saved to plaintext files in the `data_<algo>_<buffer_size>` directory:

- `curl.txt` contains the start time and duration of each web request.
- `ping.txt` is the output of a `ping` command between the two hosts. The command is configured to send a ping every 100ms, as opposed to the default 1 second.
- `qlen.txt` records the queue length of `h2`'s link, alongside a timestamp.
- `tcp_probe.txt` is the most complicated file. It records all the metadata from the iperf packets. We use this to record the TCP congestion window (`cwnd`) as seen by the kernel.

Since all these files include a timestamp with each recording, we can then align the recordings together for the plot. `plotting.py` takes as input a `data_<algo>_<buffer_size>` directory, and produces a plot encompassing all the data.

Figure 2 shows an example of a plot generated for a buffer size of 50, with the Binary Increase Congestion (`BIC`) control algorithm. Note BIC TCP is not one of the algorithms that you are asked to investigate, though your plots will have the same pieces.

The regions highlighted yellow in the background delimit the times a web request is being sent. The width of these will vary between experiments, as the web request will take longer to complete on a more congested network. There is a fixed period of 3s between the start/end of each web request, which will not be highlighted.

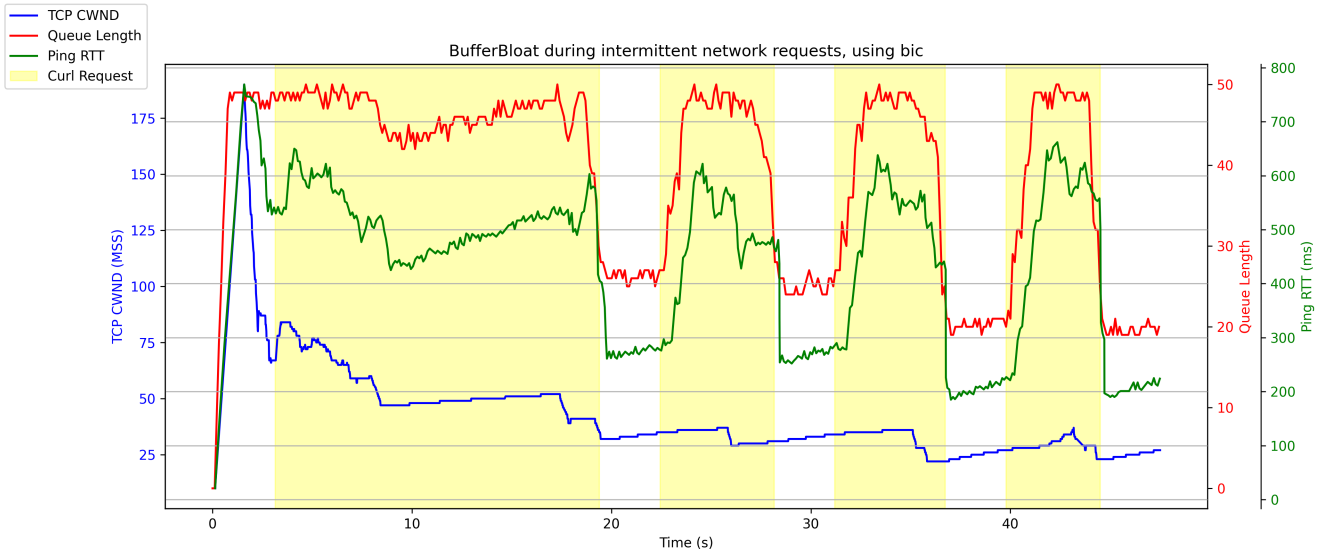

Figure 2: Behaviour of BIC TCP

The red curve corresponds to the first axis on the right. It measures the current queue length in multiples of the maximum segment size, on the output link connected to `h2` (the bottleneck as shown in Figure 1). It reflects the congestion state of our simple network. This number can never exceed the buffer size set for that experiment run, as the router drops packets that cannot be stored in the buffer.

The green curve corresponds to the far right axis. It is the `RTT` from the back-to-back ping train, giving us an estimate of latency on the network. This value cannot go lower than 20ms, since our network is defined to have that much latency in the best condition, when there is no queueing delay at all.

Lastly, the blue curve corresponds to the axis on the left. It is the current value of the TCP congestion window, which reflects how the TCP congestion control algorithm responds to the congestion state of the network. It is measured as a multiple of the maximum segment size.

Part 1: Simulator Code

To start off, you will need to finish the starter code that you will obtain after accepting the assignment on Github Classroom and cloning the repository. Most of these files are already done, though to get some practice with Mininet, you will be completing the missing portions of `main.py`. The completed Python file must be pushed to GitHub Classroom.

To start, create a class called `NetworkTopology`. This class should inherit from the Topo class, describing a topology with two hosts and one router satisfying the following requirements:

- The router connecting the two hosts is called `s1`.
- The web server `h1` has a (fast) 1Gb/s bidirectional connection to the router.
- The client `h2` has a (slow) 1.5Mb/s bidirectional connection to the router.
- The router has an output buffer on the link connecting to `h2`. The buffer size limit is determined when the network is created, so it must be passed as an argument.
- The minimum round-trip time (`RTT`) between `h1` and `h2` must be 20ms, and the propagation delay must be identical on both links.

These requirements are summarized in Figure 1.

Next, finish the average and standard deviation calculations, at the end of the `buffer_bloat_experiment` function. You will need to read the logging files to obtain this information.

Now that your code works, run `bash run.sh` in your VM to compute all the results. You should end up with several directories, each of which have a PNG plot describing the network activity for that run. Expect this to take about 8 minutes.

Submit both the `data_<algo>_<buffer_size>` directories and your completed `main.py` to Github Classroom.

Part 2: Questions

Now that you have your experimental results, answer the following questions. Provide the answers in `readme.md` and push this to GitHub Classroom.

1. What was the mean and standard deviation of webpage fetch time when using the `reno` TCP congestion algorithm for a buffer size of 10 packets? How did this result change with a buffer size of 50 packets?
2. Looking once again at the `reno` congestion algorithm, how are `RTT` measurements impacted by buffer size? Refer to the plots generated for the buffer size of 10 and 50.
3. How does the `cubic` congestion algorithm compare to `reno` in terms of webpage fetch time (a measure of throughput) and ping `RTT` (a measure of latency)? Would your answer change if the link between the router and `h2` had much higher bandwidth (**note:** this part of the question is for intellectual curiosity only and not part of the grade)?
4. Considering the buffer size of 50 packets, how does `BBR`'s performance, in terms of latency and throughput, compare to `reno` and `cubic` from your experiments? Which congestion control algorithm can better mitigate the bufferbloat problem? Explain your finding.
5. Describe in technical terms why increasing buffer size reduces performance, causing the bufferbloat effect. Be sure to explicitly reference the plots you generated and the relationship between TCP congestion control and buffer size.

You should attempt referencing specific output directories (with name format `data_<algo>_<buffer_size>`) where it benefits in your answers.

Part 3: A Lossy Network

In the previous parts, we studied how buffer size could impact TCP's various performance metrics. In this part, we will consider a network with link loss (as in wireless links). Change your `NetworkTopology` class to introduce some packet loss on the links, using the configuration options for the TCLink. Choose a reasonable value for the loss rate (e.g. 5%), as even mildly high values of loss may cause the HTTP file transfer to not finish within 60s.

Use `lossy_run.sh` to generate new plots for the network with the lossy link. Submit these output directories (named `lossy_<algo>_<buffer_size>`) into your Github Classroom repository.

Answer the following questions, referencing the output directories you are referring to for each question:

1. Considering how the TCP congestion window size changes and the queue length, do you see any difference in the plots from Part 2 and Part 3? Give a brief explanation for the result you see.

2. Considering the buffer size of 50 packets, which of the three congestion control algorithms was most performant (in terms of throughput) in the presence of increased random loss? Why do you think this is the case?

Since packet loss happens randomly on the links, you may need to run the experiment a couple of times, to get a more reliable estimate of throughput.

Submit your updated (with loss) `main.py` to Github Classroom. Do not change the name of `main.py` for Part 3; just add a comment in the updated file to indicate what has changed from Part 1.

Submission

Submit all the required files via GitHub Classroom. Do not use compressed files or archives.

The following files must be included in your repository.

1. `readme.md` containing your answers to the questions. At the very top of the readme file before any other lines, include **a header** with the following information: **your name, student ID (SID), and CCID**. For example:

```
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
# Name : Jane Doe
# SID : 1234567
# CCID : jdoe
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

2. An updated `main.py`. We should be able to `bash run.sh` and reproduce your results in the VM. **We will not accept assignments that do not work in the VM.** Submit the latest version of `main.py`, for whichever part you got to.
3. Output directories. These are the ones named `data_<algo>_<buffer_size>` and `lossy_<algo>_<buffer_size>`.

Grading

Part 1: 30% Network topology: 75%, Statistics (RTT and download time): 25%

Part 2: 50% 20% for each question. Full marks obtained by answering all parts of the question sufficiently.

Part 3: 20% Code correctness: 20%, Answering questions: 80%

The submitted output directories should be referenced in your answers.

Misc. Notes

- This assignment must be completed individually without consultation with anyone. Questions can be asked from TAs in the labs.
- The code must be written in Python.
- All code must be written in the VM image provided on Canvas. **Submissions must run in the VM, where they will be marked.** Be absolutely certain to test this.