Technical report

We utilized python and a discrete event-based simulation. Python was chosen (over other language choices such as C++ and Java) due to the familiarity of the language among all group members, as well as its ease in running quick scripts and simulations. Furthermore, Github was used for source control, as it was the most comfortable for the members of the team, and had the most ease/familiarity.
Discrete event-based simulation was also chosen, where a global priority queue of events exists, and an associated global timer. As events occur, and new events are generated, they are inserted into the priority queue. Because we utilized a priority queue, events which are expected to occur sooner rather than later are done first, regardless of the time at which they are spawned or generated. We believed that this form of simulation offered a simple but powerful form of event simulation, and thus was chosen over alternatives such as continuous and process-based simulations.

Files:

`classes.py`, which houses information about all the classes we used, including the overall network, routers, hosts, flows, and packets. This file also contains the helper and member functions of these classes.

- bufferQueue: A queue class used to keep track of buffers used in links.
- Network: Keeps track of the devices, links, and flows that make up the entire network for our simulation.
- Device: A superclass which both hosts and routers use. Contains information about links related to the host/router.
- Router: A Device which has more specific functions that deal with routing tables, links, and packets.
- Host: A Device which contains a link and deals with packet receiving/sending.
- Flow: A more complicated class which deals with congestion control (including TCP Reno and TCP Fast), generating/receiving and handling packets, and more.
- Link: A class which contains a buffer, and deals with sending packets between two locations and moderating rates.
- Packet: Contains information about the data being sent from one point to another.

`simulation.py`, which contains the event class, used to handle the various different types of events that occur throughout the simulation, including:

- Initialization of flows
- Rerouting of routing tables
- Updating window sizes for TCP-FAST
- Putting packets in link buffers
- Sending packets through links
- Receiving packets and updating accordingly
- Generating/selecting new packets
- Resending packets in the case of a dropped packet

- Also, the file contains the simulator class, which deals with the events and the global priority queue.

`runSimulation.py`, which is the actual file we invoke to run our simulation.

- Takes in information from the .json file regarding the specific test we are running, and initializes the network and relevant classes. Then, calls functions from both `classes.py` and `simulation.py` to run the simulation.

`metrics.py,` a self-contained class which visualizes the data returned from our simulations. Whenever an event is dequeued and finished processing, the current state of the network is passed into the Metrics class, which records a "snapshot" of the network. The metrics class logs the following information for specified flows and links:

- Link rate
- Link buffer occupancy
- Packets dropped by the link
- Flow rate
- Flow window size
- Flow packet delay (calculated as the round-trip-time of the last send packet)

Additionally, the data can be visualized in a few ways:

- The data logged every time an event is dequeued
- The data is logged once per small time interval
- The data is summed over a small time interval, and a time-based average within that interval is recorded. The exception to this is packet loss, which is logged as the total number of packets that occur over the interval.

`constants.py`, which contains important constant numbers that are used throughout the simulation. This helped reduce the possibility for errors that could arise from using magic numbers.

Documentation was done using Sphinx, which auto-creates documentation based off of comments made within our `.py` files. The documentation can be accessed through the file `index.html`, located in the directory `NetworkSimulator/build/html/`

Congestion Control:

Two different forms of congestion control were explored in this project: TCP Reno and TCP-Fast. Both these forms of congestion control were implemented through the use of a sliding window protocol. This protocol has a few notable features:

1. For each flow, all packets are initialized beforehand, and these packets are all placed in an array.
2. A window is represented by two indices; a lower boundary and an upper boundary. The window size is the number of packets that these bounds can capture. Since window size is not necessarily a whole number, this size is rounded down. Packets inside these bounds are the ones being operated on.
3. Every time a window's lower bound packet is received, the lower bound moves to the next unacknowledged packet in the array, and the upper bound is reset accordingly. This is the window "slide".

Packets are sent when the sliding window is over unsent packets in the packet array. Packets in the array are denoted as acknowledged through a separate boolean array, with indices corresponding to the packet index in the packet array. Packets are expected to be acknowledged in order, and a packet is dropped if three acknowledgments do not match the expected acknowledgment packet.

TCP Reno works by implementing several rules. First, two stages were implemented in the TCP Reno congestion control algorithm: Slow-Start (SS) and Congestion Avoidance (CA). Both stages update the window every time an acknowledgment is received, but the update amount is different. The update rules are:

1. Slow-Start Acknowledgement Update: New Window Size = Old Window Size + 1
2. Congestion Avoidance Acknowledgment Update:
   New Window Size = Old Window Size + (1 / Old Window Size)

Slow-Start ends once the window size exceeds a specified threshold. After the window size surpasses this threshold, the congestion avoidance stage takes over. At the start of the simulation, the flow will initially start at Slow-Start with no threshold. Once a packet is dropped, TCP Reno will halve the window size and undergo Fast Retransmit, where the dropped packet is resent. If the flow was currently in Slow-Start, then the threshold is updated to be the updated window size, and the flow proceeds to enter into congestion avoidance. Whenever a timeout is detected (in which a packet acknowledgment has not been received after a significant period of time), the flow re-enters the Slow-Start stage to re-establish a threshold.

TCP-Fast works by updating the window size every two RTTs, as calculated using the packet travel time. Furthermore, it follows the expression:

$$W(t) = W(t+1) = \frac{RTT_{min}}{RTT} W(t) + \alpha$$

for alpha = 20. Since TCP-Fast is delay-based algorithm, it can generally maintain a more constant window size, avoiding the oscillations which are evident in loss-based algorithms. Window size and boundaries are thus adjusted periodically using this algorithm, which does not really consider packet drops directly, but instead takes this into account by utilizing the last RTT.

Routing:

At the beginning of the simulation, each router initializes its router table to be adjacent neighbors. It then floods its neighbors with its current routing table. Upon receiving a routing packet through a given link (link) from another router (other), each router (self) follows the following pseudocode

```
for each device in routing_packet:
    if(dist[device][self] > dist[device][other] + link.latency):
        updated routing table with this shorter distance and corresponding link
if the table was changed:
    send updates to adjacent neighbors
```

The simulation calculates new routing tables every 5 seconds (dynamic routing). The main difference is that link.latency is estimated through the length of the link buffer and the link rate.

Running the Simulation:

The simulation is run on runSimulation.py. The two required arguments are

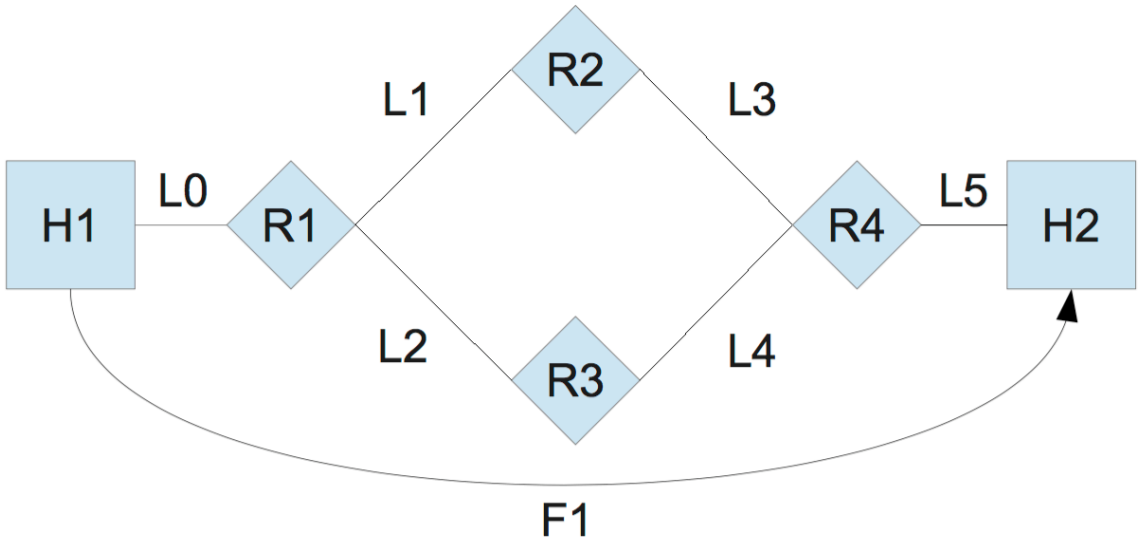
- -j JSON_FILE_NAME, which takes in an json input to initialize the network
- (--Reno | --FAST), which specifies the type of congestion control used.

There are various optional arguments that can be given to the command line, of which more information can be found by entering the command
`python runSimulation.py -h`
Example commands are as follows:

- Test Case 0:
  - `python runSimulation.py -j test0.json --FAST -m --avg -l L1 -f F1`
- Test Case 1:
  - `python runSimulation.py -j test1.json --FAST -m --avg -l L1 L2 L3 L4 -f F1`
- Test Case 2:
  - `python runSimulation.py -j test2.json --FAST -m --avg -l L1 L2 L3 -f F1 F2 F3`

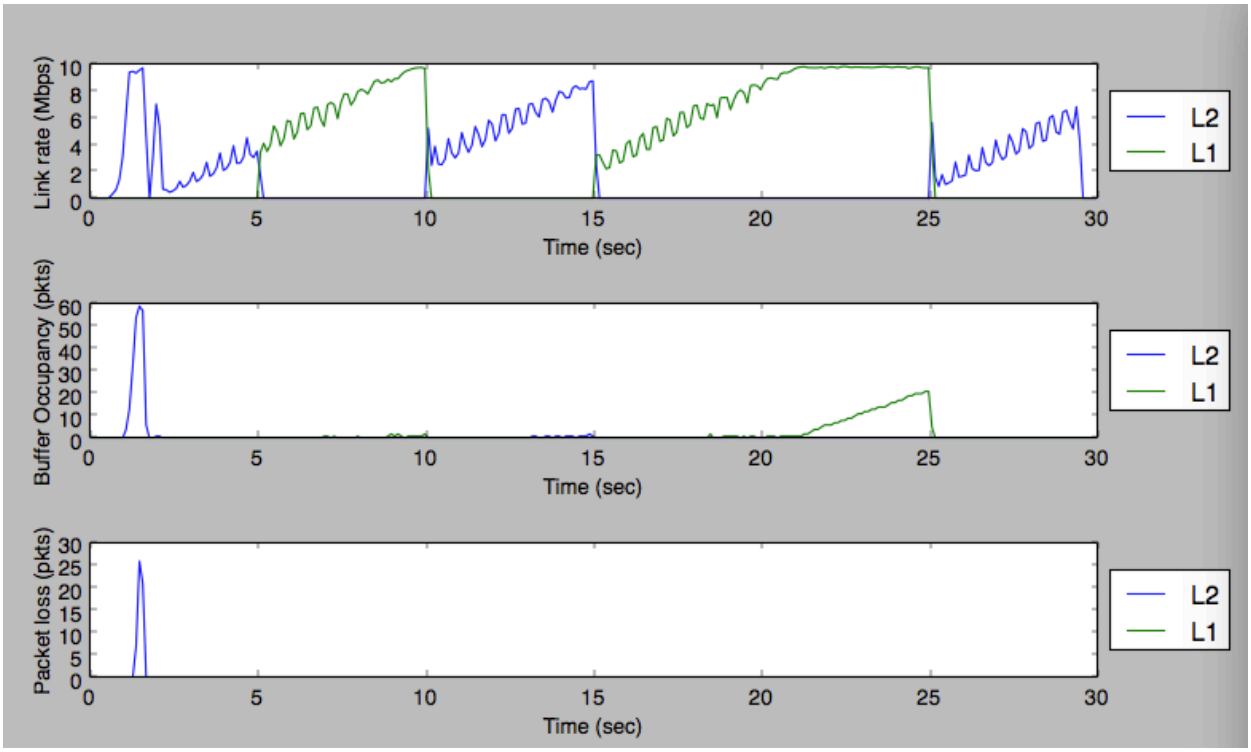
Results & Analysis:

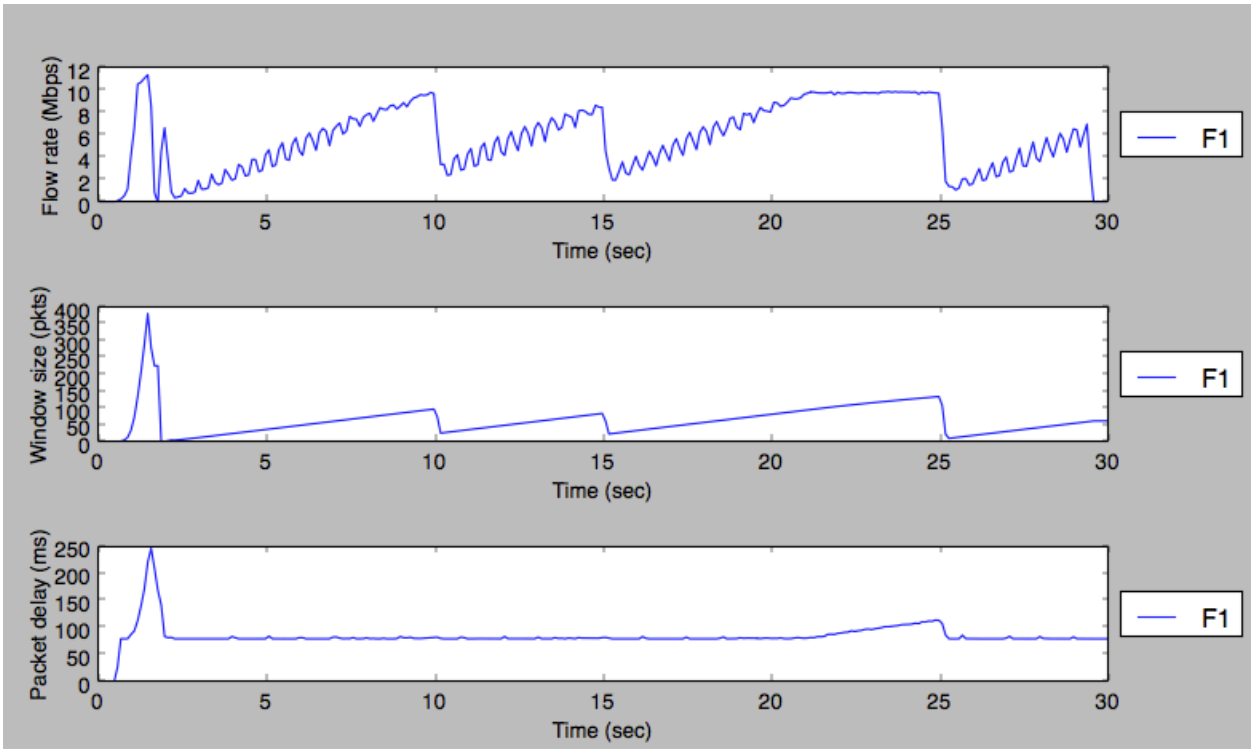
The network configuration for test case 1 is given as follows:



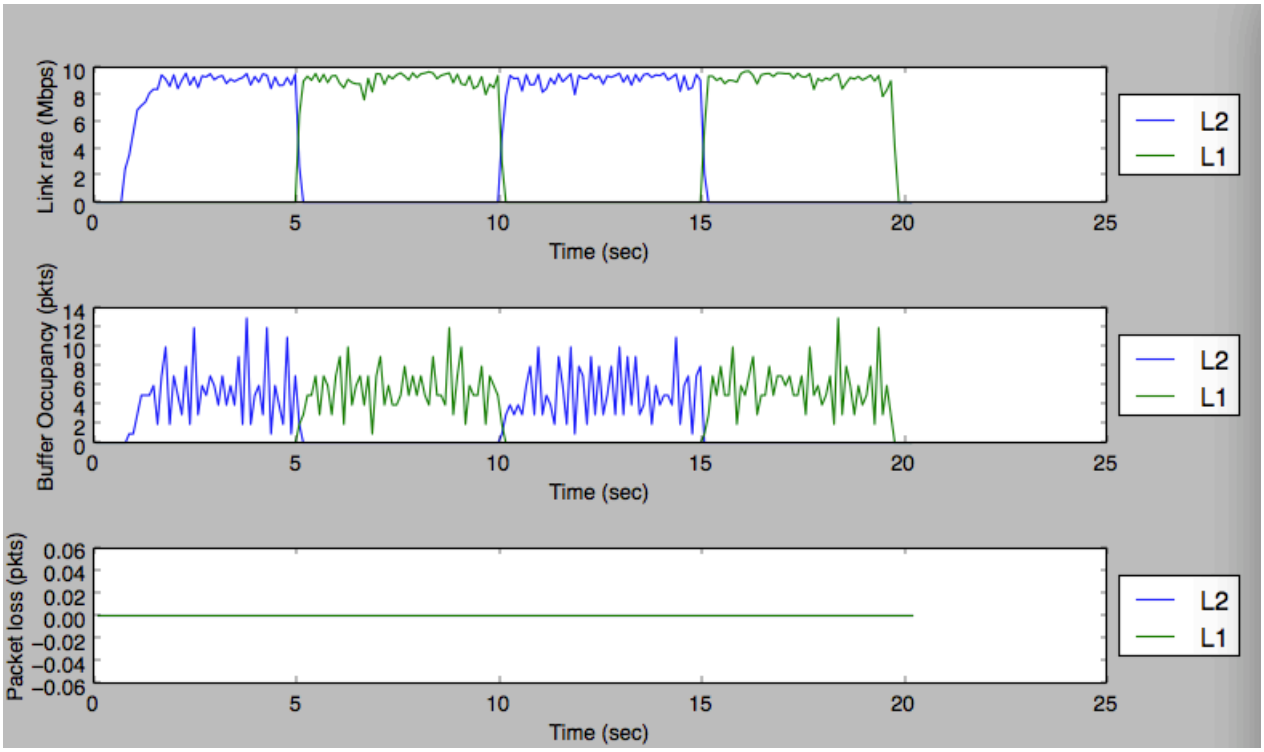
Base image taken from the description of test cases on the CS/EE 143 website, http://courses.cms.caltech.edu/cs143/

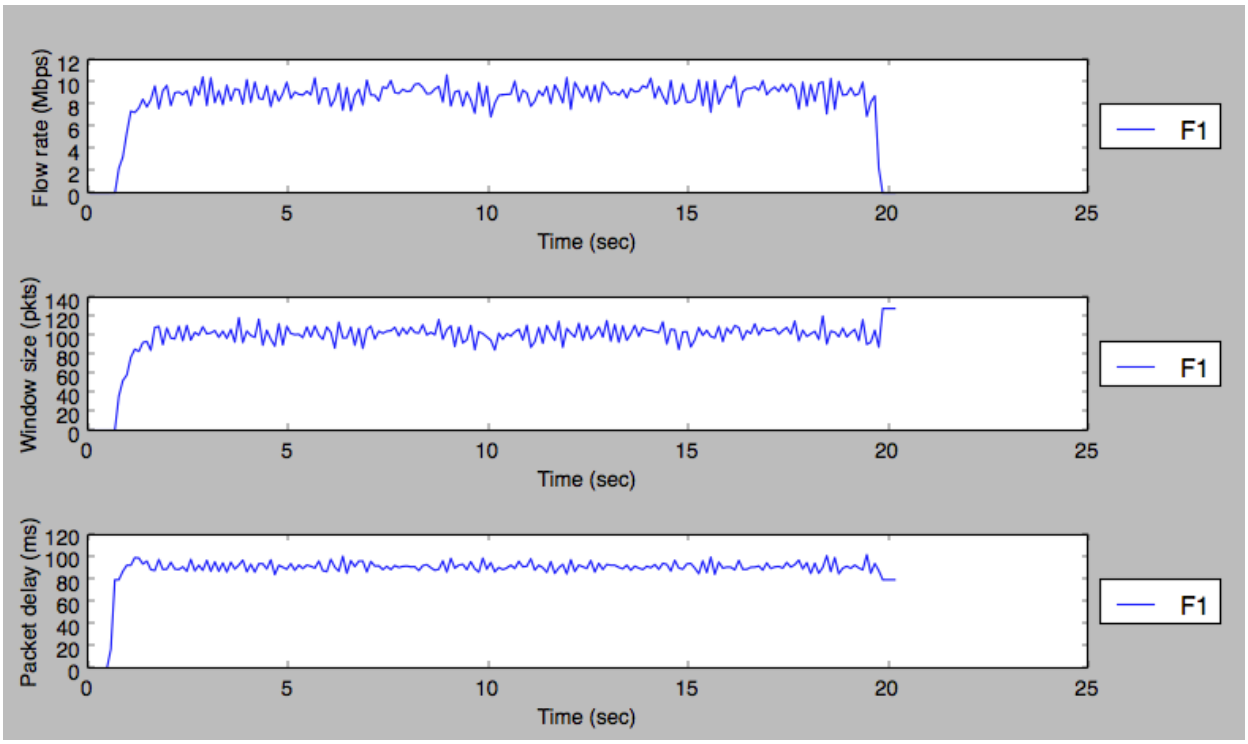
Our network simulation yields the following output, using Reno congestion control:



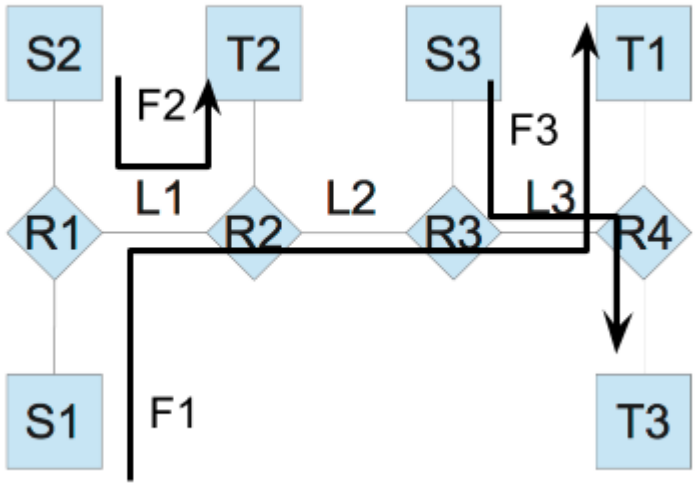


And using FAST congestion control:





The network configuration for test case 2 is given as follows:



Base image taken from the description of test cases on the CS/EE 143 website, http://courses.cms.caltech.edu/cs143/

With the following specifications:

Link Specifications

	Link rate (Mbps)	Link delay (ms)	Link Buffer (KB)
L1, L2, L3	10	10	128
Other links	12.5	10	128

Flow Specifications

	Flow Src	Flow Dest	Data Amt (MB)	Flow start(s)
F1	S1	T1	35	0.5
F2	S2	T2	15	10
F3	S3	T3	30	20

TCP-FAST theoretical analysis

It is assumed that all flows in this network follow a TCP FAST congestion protocol

The total amount of data sent by a flow during a time interval is

$$time \times \frac{R}{1+p}$$

where R is the amount of data sent in 1 second, and p is the propagation delay.
The steady state window size is given by solving

$$W(t) = W(t+1) = \frac{RTT_{min}}{RTT} W(t) + \alpha$$

which yields

$$W(t) = \frac{\alpha}{RTT - RTT_{min}}$$

where RTT is the round trip time of a packet. Here it is assumed that alpha is 20.
Flow rate of flow *i* is given by

$$x_i = \frac{\alpha}{p_j - q}$$

where p_j denotes the queue delay from link *j* (the bottleneck link for flow *i*) and *q* denotes the queuing delay of flows that were previously started.
Between 0.5 and 10s, only F1 is running, and it is free to use all the capacity in the links. Since L1/L2/L3 have the minimum link rates in the path from S1 to T1, the flow rate is just 10Mbps.

Additionally, all packets will be buffered at L1, and therefore L2 and L3 will have zero queuing delay. F1's theoretical flow rate should be 10Mbps, and the expected amount of data sent is 11.2 MB.

Between 10s and 20s, F2 enters the network, and F1, F2 will share L1. Assuming that F2 calculates its packet RTT to include queue delay of L1, the maximization problem is written as:

$$x_1 + x_2 = \frac{\alpha}{p_1} + \frac{\alpha}{p_1 - q} = 10Mbps = 1280\frac{packets}{s}$$

which yields F1 rate to be 3.79 Mbps,, and F2 rate to be 6.21 Mbps.
During this time, F1 will have sent 4.46 MB, and F2 will have sent 7.89 MB.

After 20s, all three flows are running. F1, F2 still share L1, and F1, F3 share L3.
Since packets are buffered at L1, L2 and L3 are underutilized, and therefore there is no queue delay at L3. Thus, the maximization problem is

$$\frac{\alpha}{p_1 + p_3} + \frac{\alpha}{p_1 - q} = 1280\frac{packets}{sec}$$

$$\frac{\alpha}{p_1 + p_3} + \frac{\alpha}{p_3} = 1280\frac{packets}{sec}$$

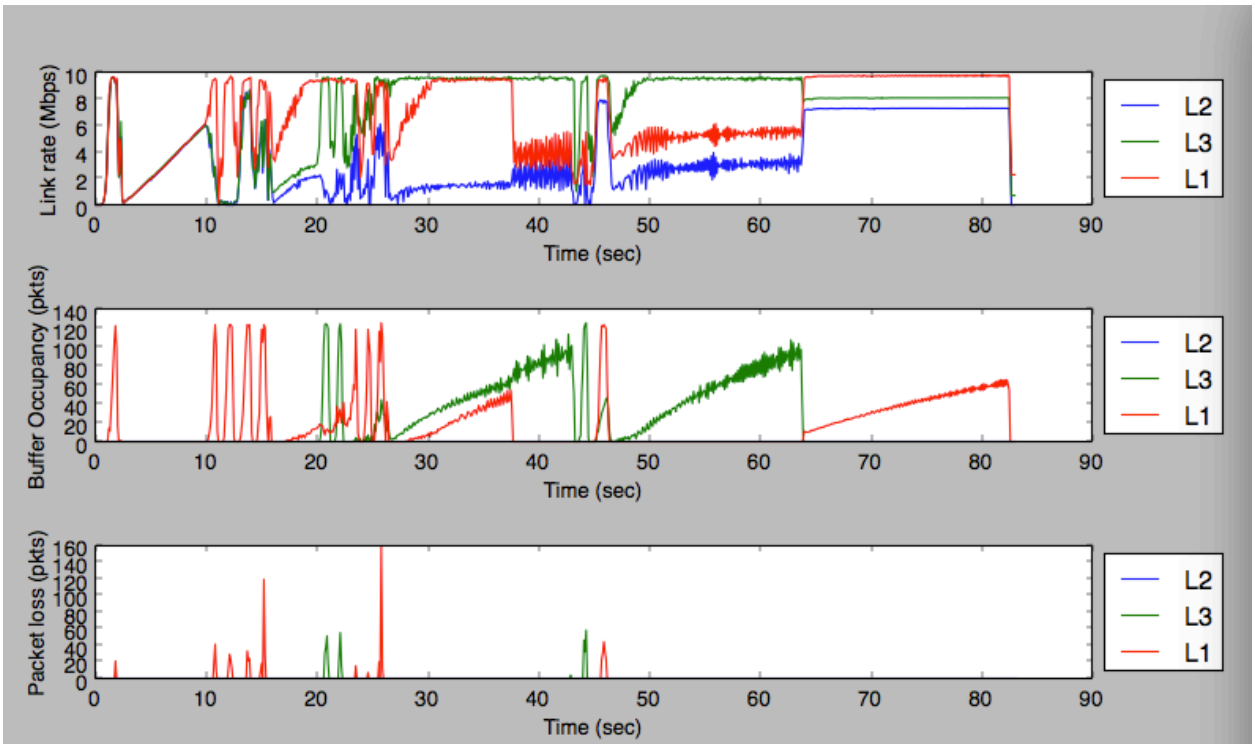
Solving this yields F1 rate to be 2.68Mbps, and F2 rate = F3 rate = 7.43Mbps.

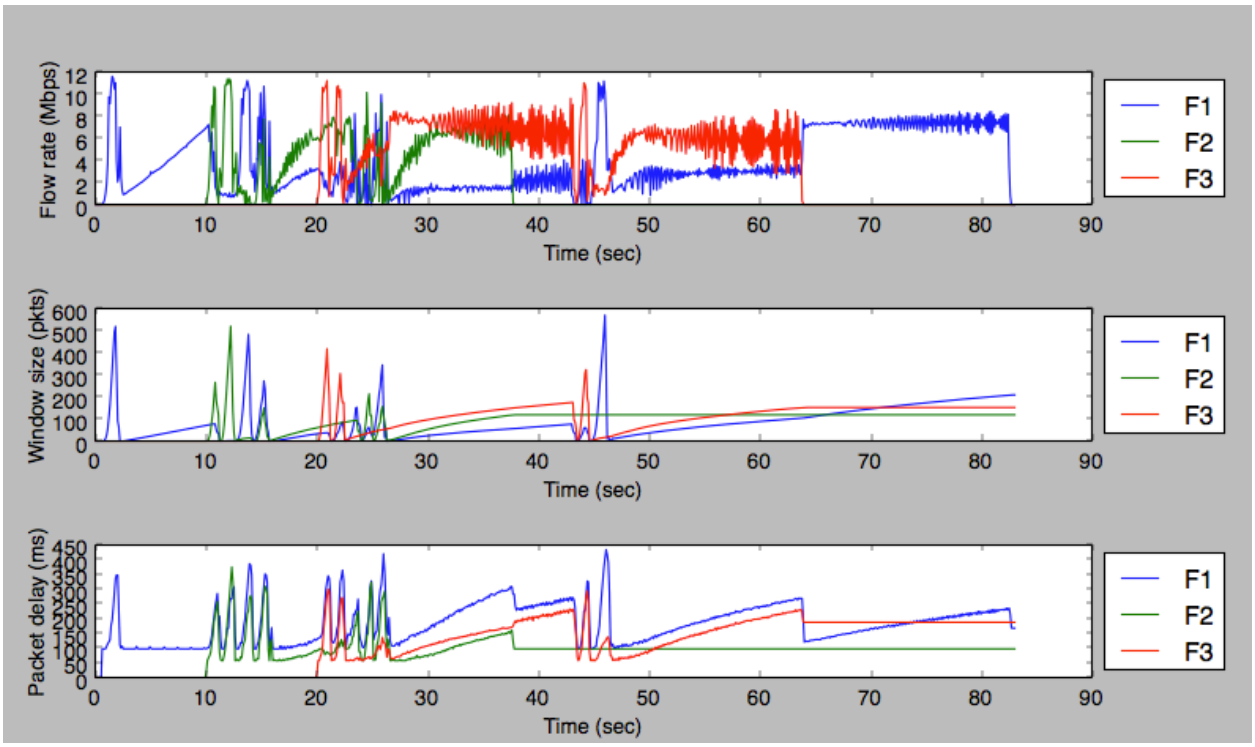
Since F2 has 7.11MB remaining to send, the time required is 8.66s. During this time, F1 will have sent 2.73 MB, and F3 will have sent 7.11MB.

Thus, after 28.6s, F2 finishes, and only F1 and F3 remain in the network. Since there is no queue delay at L2 or L3, the flows share L3 equally: F1 rate = 5Mbps, and F3 rate = 5Mbps. F1 only requires 16.61MB, whereas F3 requires 22.89MB, so F1 will finish in 26.5 seconds. During this time, F3 will have sent 16.61MB as well.

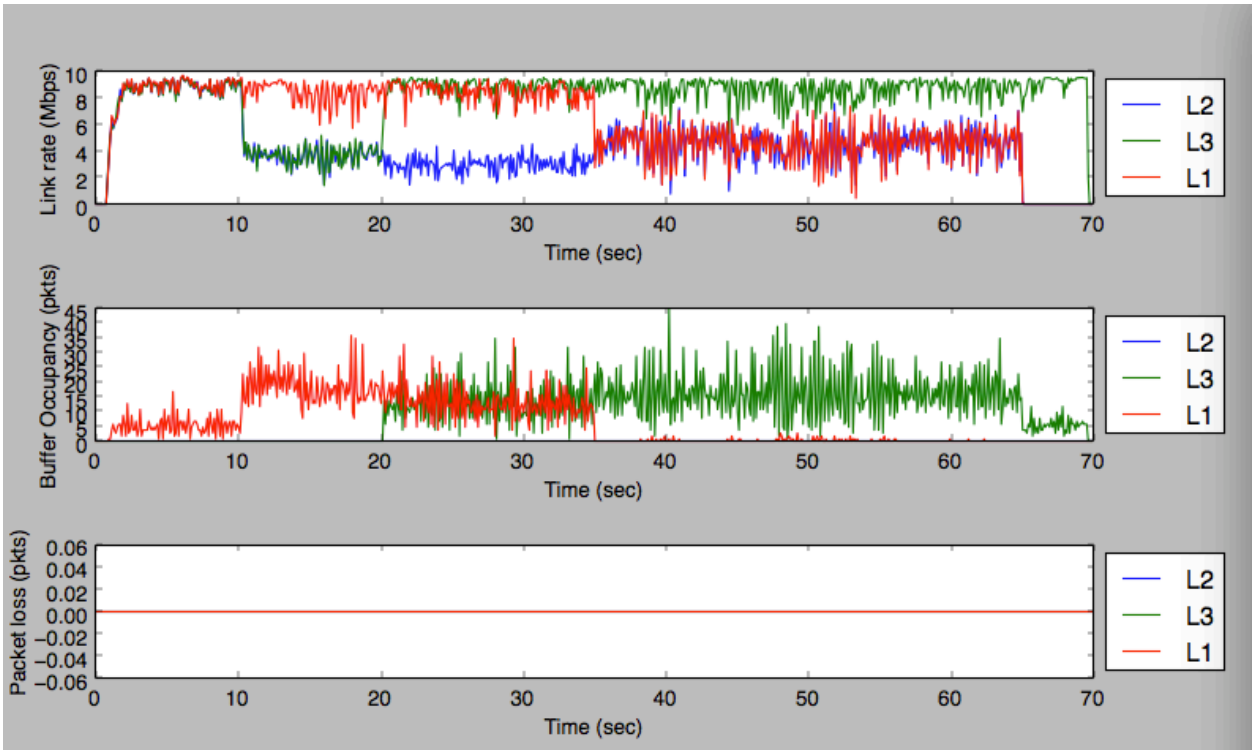
After 55.1s, F3 still has 6.28MB remaining to send, and since there are no more flows running in the network, this takes 5.12s, and the simulation completes at 60.22s.

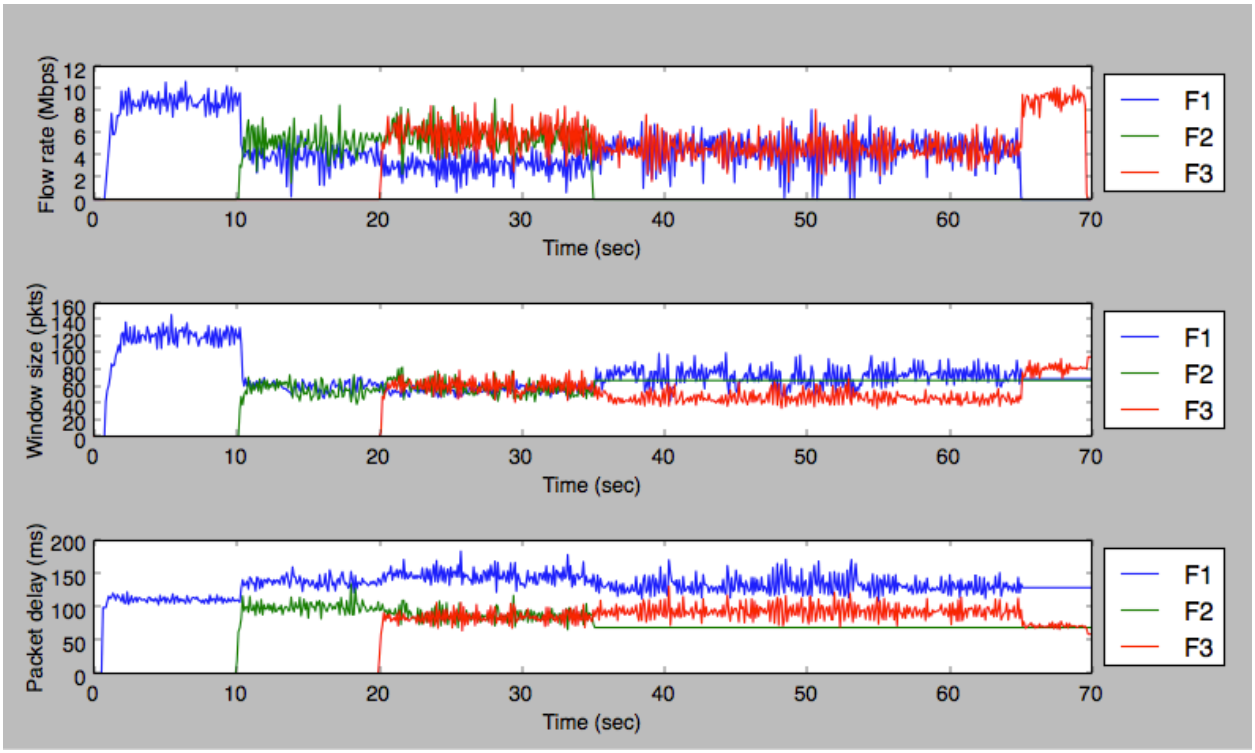
Our network simulation yielded the following data output, using Reno congestion control:





And, using the FAST congestion control:





TCP-FAST Simulation Analysis

For test case 1, the results are mostly in agreement with expected output. Since there is only 1 flow in the network, the flow rate will just increase until its reaches the maximum bottleneck capacity (10Mbps).

From 0.5 to 10s, flow rate for F1 hovers around 10 Mbps. At time 10s, F2 enters the network, and F1 and F2 have rates of about 4 and 6 Mbps, respectively. At time 20s, F3 enters the network. F1 decreases to about 3Mbps, whereas F2, F3 have approximately the same rate, about 7 Mbps. F2 ends slightly later than expected (it takes around 25s to finish, which is 6s later the theoretical calculation), after which F1 and F3 have the same rate of about 5Mbps. F1 continues until about 65s (takes 30s to finish, which is 3.5s later than the theoretical calculation). Once F1 leaves the network, F3's flow rate increases back to 10Mbps as expected, and takes around 5s to complete, which is very close to the theoretical calculation.

Thus, the results gained from the simulation are mostly in agreement with the expected theoretical output.

TCP-Reno Simulation Analysis

As you can see in Test Case 1's and Test Case 2's graphs above, TCP Reno's update protocol can be seen clearly between Slow Start and Congestion Avoidance. Right when the simulation started, the window size and link and flow rates spike dramatically. This is because window size doubles for every round trip time, due to the slow start window update protocol. This spike is quickly followed by a sudden drop in rate and window size; this represents dropped packets caused by too large of a window size. It is at this point where the window size is halved and a slow start threshold is established (which is half the window size). Theoretically, any time the window size is above this slow start threshold, the update protocol switches from slow start to congestion avoidance (window size increases by 1 for every window of packets acknowledged).

We see in Test Case 1 graphs above that congestion avoidance begins with a slow start threshold at 0, though it was anticipated for the slow start threshold to be larger than that (around half the window size peak of slow start). The reason for this is because of Time Out events; in the case of a Time Out event, the host expects a packet but hasn't received it after significant time. Slow start protocol starts after a time out event is detected. After lots of debugging, we also see that a dropped packet occurs shortly after the Time Out event, effectively initializing the Slow Start threshold to be very small. This is why the window size updates in accordance with Congestion Avoidance shortly after the first Slow Start phase.

We also see a trend where the buffer occupancy increases linearly as the link rate is full. This is because of how our sliding window protocol works; the moment a packet is correctly acknowledged and received, the flow sends another packet. Therefore, the buffer size would increase linearly, as packets are usually only sent out one at a time during congestion avoidance. This also explains why the link rates occasionally increase linearly instead of exponentially; after the Slow-Start phase, packets are usually only sent out one at a time, because the window size doesn't increase fast enough to warrant more packets to be sent out for each acknowledged packet. This is why you see a linear trend in Test Case 1 and for some instances in Test Case 2.

Project process:

The project process offered numerous challenges, as well as numerous learning opportunities. For many of us, this was our first significant group project, and thus the idea of version control was an unexpected challenge alongside the typical challenge of designing, writing, and completing a product.
Given these two initial hurdles, the first task was to become familiar with version control. Several branches were opened, and once we had a better understanding of merging, we then approached the project together, such that a basic framework for the simulator could be reviewed by each team member. We established milestones and goals, and helped teach each other how to fit various pieces of the code together. Thus, the first working baseline code of the following was established on one branch:

- The classes representing flows, links, hosts, routers, packets
- The simulator containing a global priority queue of events
- The class for parsing a command, and initializing the network

Once test case 0 was completed, our project advanced into the deeper topics of the project. Because test cases 1 and 2 were significantly more sophisticated, work was divided up, and more branches were created with different people primarily working on different aspects of the network simulation.

- Albert Ge worked on data visualization and metrics
- John Li worked on TCP-Reno and sliding window protocol
- Jonathan Joo worked on TCP-Fast and documentation
- Matthew Jin worked on routing (static and dynamic)
- Shared duties involved architecture design, presentation preparation, debugging, etc.

While debugging was done mostly independently, often merging branches would reveal more issues, which could be either bugs completely within one member's code, or a result of different design philosophies. To resolve this, merging and subsequent debugging was often done together via Skype or in person. Because branches were merged one at a time, careful consideration of the various branches allowed for efficient progress across multiple fronts simultaneously, while minimizing the pain of lost work. Thus, great importance was placed upon understanding the entirety (or nearly the entirety) of our working codebase, such that debugging and testing could be done by any individual in the group.
Understanding the current status of the codebase, and knowing what worked and didn't work was also useful because team members' availability in schedule were scattered throughout the week. At times, building on top of multiple members' codebases had to be started independently; thus, we learned that it was imperative to communicate with each other about the current state of implementation; in this, opening and closing pull requests and issues was immensely helpful in keeping of track of which bugs still persisted.

Overall, the project was a formative experience which allowed the members of the group to gain experience in not only applying networking concepts learned through classwork and lecture, but also to gain experience working together as a group.