Chanukya Malla
Nand Kishore
Avishek Dutta

CS 143 Final Report

Architecture Overview

For this project, our group designed and implemented an abstract network simulator. Since a given network has many components such as hosts, routers, links, flows, and packets, we decided it would be best to use an object oriented language, such as Java to better encapsulate these objects. Furthermore, our group decided that the simplest way to simulate a network would be to iterate over a priority queue of events ordered chronologically by time. These events would contain the majority of the logic necessary to perform the simulation and would add additional future events to the 'global simulation loop' upon completion. Lastly, in order to set everything up properly, the priority queue would be manually populated with initial events.

The major components in our implementation are the host, router, link, flow, and packet classes. These components contain information about themselves and the network, but contain minimal logic with regards to the simulation of the network. The host for example contains variables for data collection, a list of current flows, a list of adjacent links, and functions that are called when packets are received at a host. The routing class is relatively short and only contains hash maps for the routing table, and extra hash maps used for dynamically recalculating the routing table on certain intervals. The link class is a bit more involved; it contains variables to store congestion metrics, variables for storing buffer occupancy, and variables that represent the buffers on both ends of the link. The flow class contains information about its source and destination, start time, number of packets, round trip time, and RTT timeout. These variables are then used by our congestion control algorithms to control the window size. Finally, the packet class contains information about its size, source, and destination as well information about what type of packet it is.

Event classes are another major component of our implementation. These events are either generated externally by the user or internally by components in the network and are the main driving force of the network simulator. As mentioned earlier, they are stored in chronological order in an event priority queue. The major event types are initialize flow, routing, send packet, buffer-to-link, receive packet, negack, and collect data. The initialize flow event sets up the basic parameters for a new flow and sends out the initial packets. The routing event handles the logic for starting the dynamic shortest path algorithm to calculate the routing table for all routers. The send packet event handles the logic of placing a packet onto the correct link buffer.

The buffer-to-link event removes the packet from the buffer and schedules an event for the packet to arrive at the destination after a delay. If there are more packets in the buffer, we schedule a buffer-to-link event for the next packet. The receive packet event handles the logic of receiving a packet at a host or a router. If a data/acknowledgement packet is received then it is either processed if the current component is a host or is routed if the current component is a

router. If a routing packet is received at a router, we update the routing table using the bellman ford algorithm.

A negack event is triggered when the acknowledgement for a packet is not received by the end of the RTT timeout period. The event is potentially rescheduled based on the most recent round trip time estimate. Otherwise, if the packet is still in the sending buffer, then the packet is lost. Then, based on which TCP algorithm is being run, we adjust the window size accordingly.

Finally we have the collect data event. This event is triggered on a specific interval and collects data over the links (occupancy data, packets lost, flow rate), flows (send rate, receive rate, round trip time, window size), and hosts (send rate and receive rate).

Assumptions

We made a few assumptions when implementing our network. Namely, these were that all routers have knowledge of when a routing event begins, that RTT timeout of 3* current RTT is a reasonable estimate, that the receiver has an infinite buffer size (hence making flow control unnecessary), and that links are full duplex and are first in first out. Assuming that all routers have knowledge of when a routing event begins allows us to simplify the dynamic shortest path algorithm to update routing tables and is a reasonable assumption because it guarantees that our algorithm convergences. Our estimate for RTT timeout allowed us to create negack events without the need for complicated algorithms while still approximating real world behavior. Having an infinite buffer size for the receiving host is not a very realistic assumption, but since our guidelines said this to be true, we were able to avoid having to implement flow control. Finally, we decided to implement full-duplex links rather than half-duplex in order to eliminate some of the errors we were encountering with half-duplex and is reasonable since many links are in fact full-duplex.

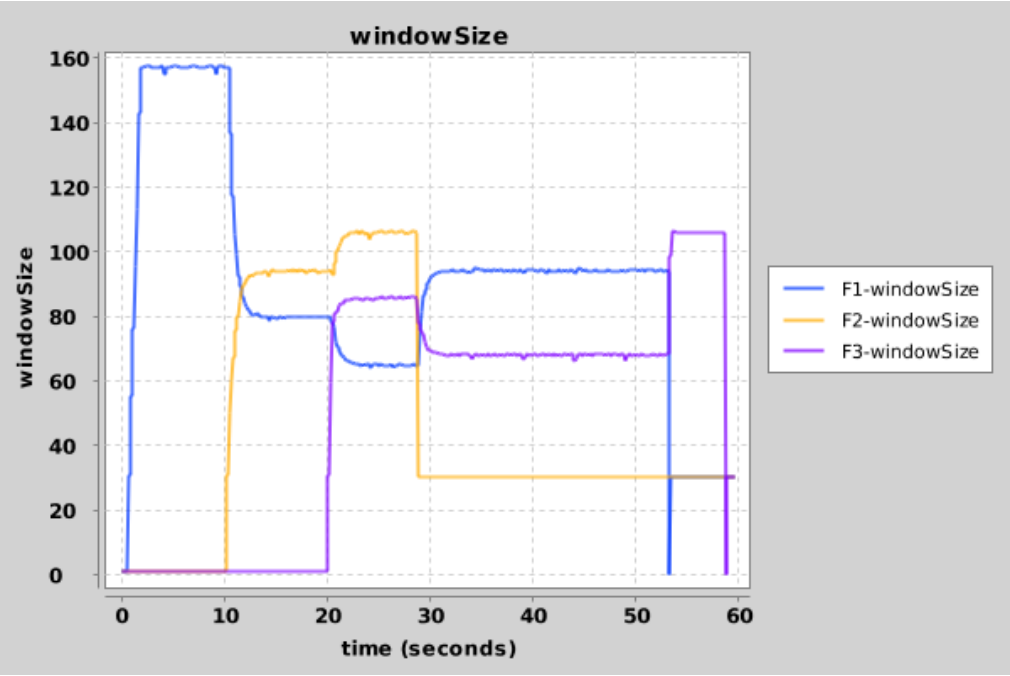
Test Case Analysis

Running test case zero through our implementation using TCP Reno and FAST-TCP gave us expected results. For TCP Reno, we observed a shark-fin like graph for window size, buffer occupancy, and average packet delay. This is indicative of the typical congestion avoidance phase followed by a fast recovery/fast retransmit phase that is common in TCP Reno. Additionally, the observed flow/link rates were primarily constant except for occasional dips which were presumably caused by the fast recover/fast retransmit phase. For FAST-TCP, we observed that the graphs for window size, buffer occupancy, and packet delay stayed constant over the course of the simulation. This, coupled with the fact that packet loss was zero, is in line with the stability that is expected from FAST-TCP. Plots from this test case can be found in the plots folder of our github repository.

Test case one also gave us results that were expected. For our simulation, we timed the dynamic shortest paths algorithm to run every five seconds which resulted in an oscillatory pattern for our link rates and buffer occupancies. This oscillatory pattern was caused by the structure of the network; there are two paths from hosts H1 to H2 and when one gets congested, the other is preferred. Although this oscillation was observable when using TCP Reno, it was more salient when using FAST-TCP. Plots from this test case can be found in the plots folder of our github repository.

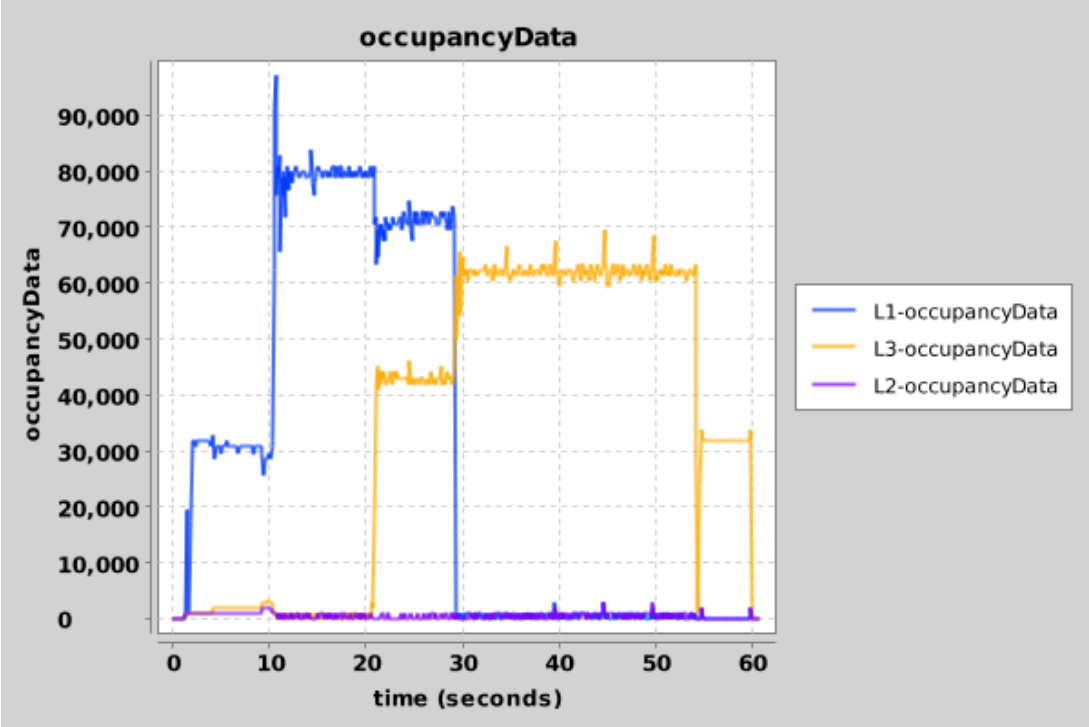
Running test case two through our network simulator using TCP Reno also gave us expected results. Although some of our plots were a slightly messy, we were able to notice distinct changes in behavior at the ten and twenty second marks which indicates that our network was reacting to additional flows being introduced into the network.

Test case two with FAST TCP ran and gave results that also mirrored our expectations. The window size for flow 1 ramped up quickly before hitting a plateau. Once flow two starts, the window size for flow 2 quickly grows while flow 1's window size gets cut in half such that flow 1 and 2 have almost equivalent window size. Then finally when flow 3 starts, flow 1 experiences another slight decrease in window size while flow 2 enjoys a slight increase in window size and flow 3 plateaus approximately half way between flow 1 and 2. Then at approximately 30 seconds into the simulation, flow 2's window size crashes to a third of what it was while flow 1 has a slight increase and flow 3 has a slight decrease. Once flow 1 finishes, it crashes to 0, allowing flow 3's window size to grow but flow two remains constant.

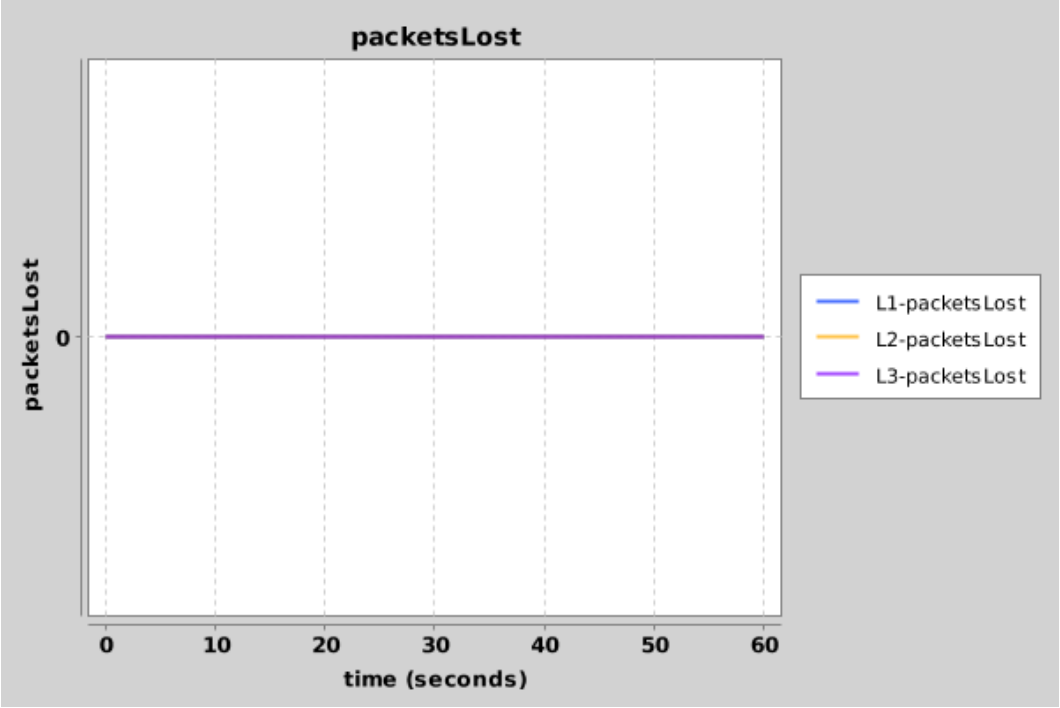


The buffer occupancy data also behaved as expected. The buffer for link 1 grew quickly and plateaued at ~30,000 bytes/s before growing quickly again until plateauing at ~80,000

bytes/s. Then, link 3 starts receiving packets, which bottlenecks the system. Accordingly, link 1's buffer occupancy reduces to ~71,000 bytes/s while link 3 grows to ~42,000 bytes/s. Once flow 2 finishes the queueing delay at link 1 drops to 0 bytes/s and link 3's buffer grows to ~60,000 bytes/s. This trend can be observed in the diagram below.

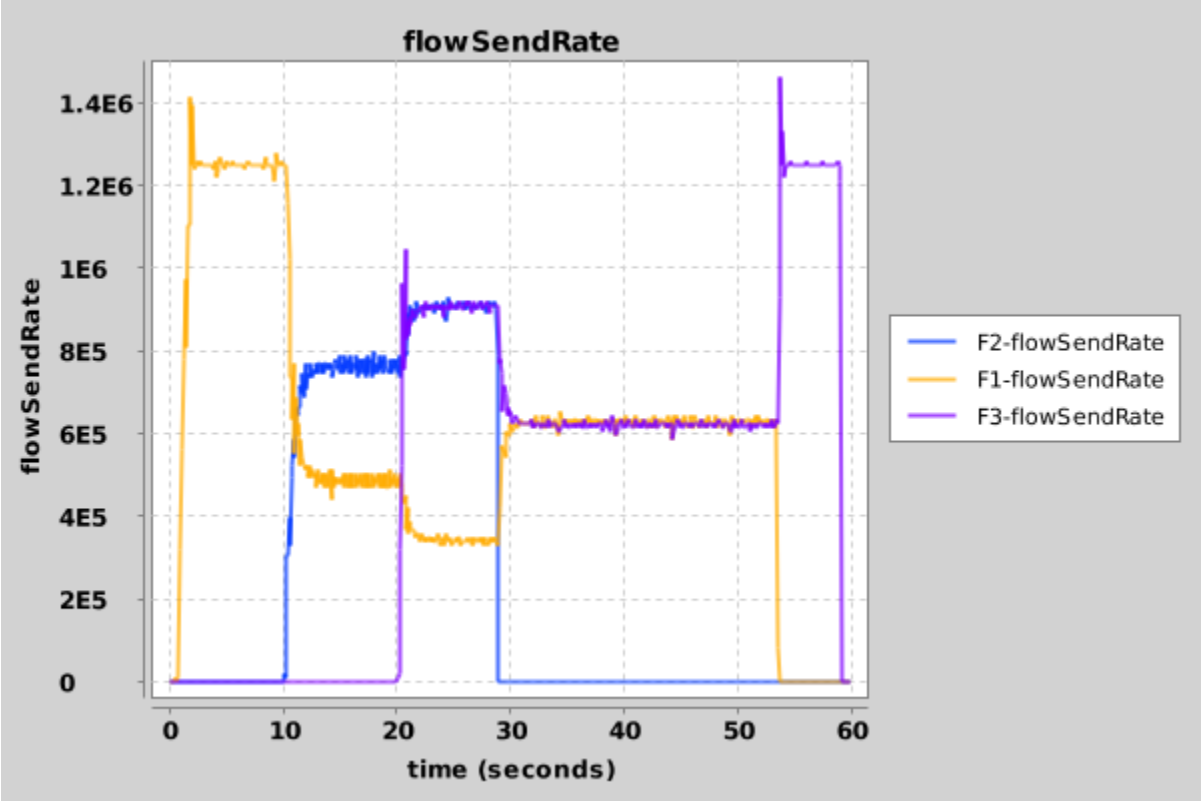


Since there was no packet loss in this test case, our data for packet loss is just 0 for all 3 links.



Chanukya Malla
Nand Kishore
Avishek Dutta

Lastly, flow rate for flow 1 grew rapidly till hitting a plateau at ~1.25*10^6 bytes/s which is to be expected since that is the link rate. Then, once flow 2 starts at 10 seconds, flow 1's rate drops to approximately ~4.7*10^5 bytes/s and flow 2's rate grows to about ~7.7*10^5 bytes/s. Then at 20 seconds, flow 3 begins, growing its rate rapidly to slightly above ~9*10^5 bytes/s, causing a slight dip in flow rate for flow 1 to ~3.4*10^5 bytes/s but a slight increase in flow rate for flow 2 to ~9*10^5 bytes/s. Note that at this time, the flow rates of flow 2 and 3 are equal. Then at 30 seconds, flow 2 crashes to 0 since the flow is finished and flow 3 and 1 converge at approximately ~6*10^5 bytes/s until 53 seconds at which point flow 1 finishes and flow 3 reaches a peak at ~1.25*10^6 bytes/s for a few seconds until it also finishes at about 60 seconds.



To summarize, here are the actual vs theoretical flow rate results for this test case using FAST-TCP as the congestion control algorithm with parameter alpha = 30 (adapted from HW 3). All values are in bytes/s.

	Flow 1 Actual	Flow 1 Theoretical	Flow 2 Actual	Flow 2 Theoretical	Flow 3 Actual	Flow 3 Theoretical
0-10s	1.25*10^6	1.25*10^6				
10-20s	4.7*10^5	4.77*10^5	7.7*10^5	7.74*10^5		
20-30s	3.4*10^5	3.34*10^5	9*10^5	9.14*10^5	9*10^5	9.14*10^5

Lastly, here are the actual vs theoretical queue length for this test case using FAST-TCP with parameter alpha = 30 (again, adapted from HW3). Again, all values are in bytes.

	Link 1 Actual	Link 1 Theoretical	Link 3 Actual	Link 3 Theoretical
0-10s	3*10^4	3.07*10^4		
10-20s	8*10^4	8.04*10^4		
20-30s	7.1*10^4	7.28*10^4	4.2*10^4	4.20*10^4

Conclusion

In conclusion, we successfully implemented a network simulator that closely modeled the given test cases. We found that implementing the simulator iteratively by focusing on one feature at a time allowed us to write fast and bug-free code. It was certainly a challenging but rewarding experience.