David Pena, Eugene Bulkin,
Catherine Jamshidi, Archana Chintalacharuvu

CS 143 Report

Overview

The simulation was written in Python, using an event-driven simulation method and storing flow specifications in XML. We assumed that hosts had an unlimited buffer size and that hosts could have at most one flow sending data, as for practical purposes hosts can send out orders of magnitude more data than the links could handle, and no test cases required more than one flow per host. For the purposes of routing, we used the Bellman-Ford algorithm to calculate decentralized routing tables, under the assumption that routers would all be able to simultaneously update routing tables every 3 seconds, so as to alleviate the need for sending packets to routers for the purpose of reminding them to start rerouting, which would add more complexity to the algorithm and would cause the algorithm to take longer to converge. In order to provide true half-duplex links, each link buffer is aware of the packets on the other buffer on that link, in order to determine which packet should be sent, again so that we do not have inconsistent results due to needing packets or other methods of information passing move between link buffers. This communication between the buffers is not very interesting and would add unnecessary complexity. The simulation supported three congestion control methods: TCP Tahoe, TCP Reno and FAST-TCP.

Approach and Architecture

In terms of implementation, we used four main classes to denote network components. The Network class was used to collect all routers, hosts and links and to trigger event handling. The Host and Router classes were used for hosts and routers, respectively, and inherited from a base Node class, which supported sending and receiving packets. The Link class was used to represent a link, and would store a LinkBuffer instance to hold onto the buffers.

In order to run the simulation using events, an EventDispatcher class was used, which would listen to any event-emitting class (Nodes, Links, and the Network all can emit events). When an event was fired with a certain time, it would be stored in the event queue. In the network event loop, at time t, the EventDispatcher would execute every event that was to be executed at or before time t, in the order they would have been submitted. The simulator ran until there were no more events in the queue. Some events were executed on a timer that was separate from the event dispatcher. These timers did not keep the simulation from terminating; they were separate from the event queue.

Hosts store all flow data internally. They emit an event to start the flow at the flow's start time, and from there we use the sliding window protocol to handle sending data. We use Go-Back-N ARQ to determine how many packets to send and to handle retransmission. Congestion control is abstracted away by storing a congestion control Protocol instance in the host depending on what the flow specified. Any time a packet is sent, received or a timeout is triggered, Go-Back-N ARQ is handled and the congestion control protocol is then notified to handle those cases if need be.

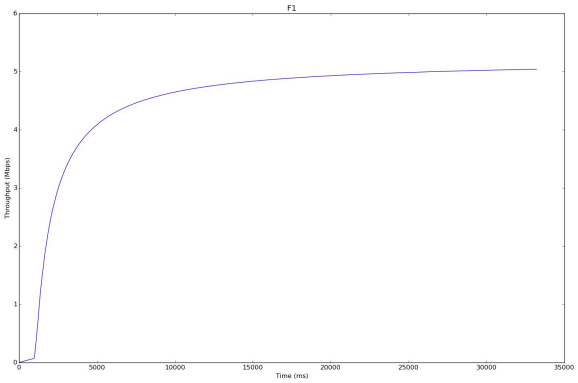
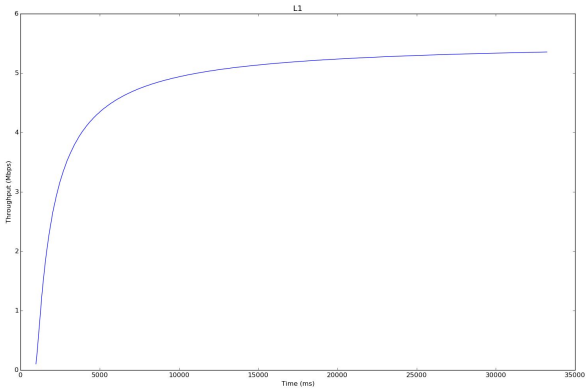
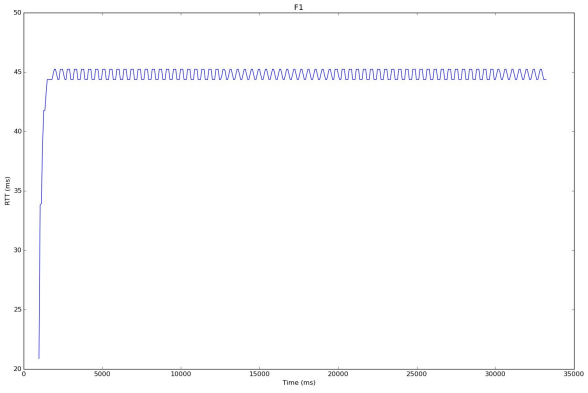
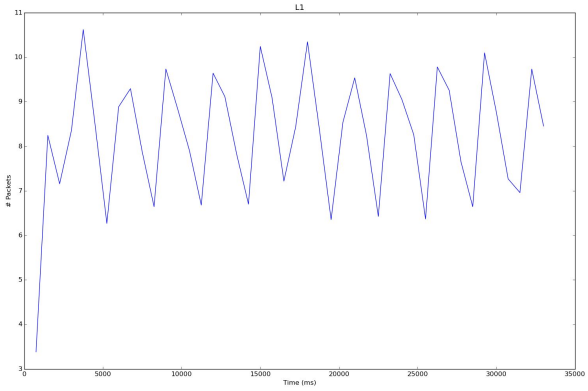
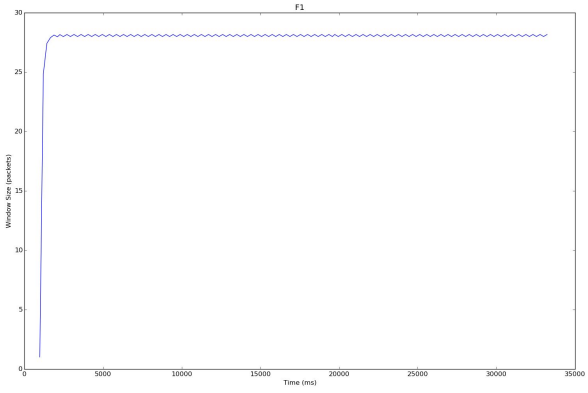
Routers all simultaneously create their routing table at time t = 0 using static costs, which are just the link rate of a given link. Dynamic cost is determined by adding average buffer time to static cost. Average buffer time is calculated by storing arrival times in a dictionary and calculating the difference between arrival and departure times from the buffer. A running average is kept and reset after the dynamic routing algorithm has completed.

Process

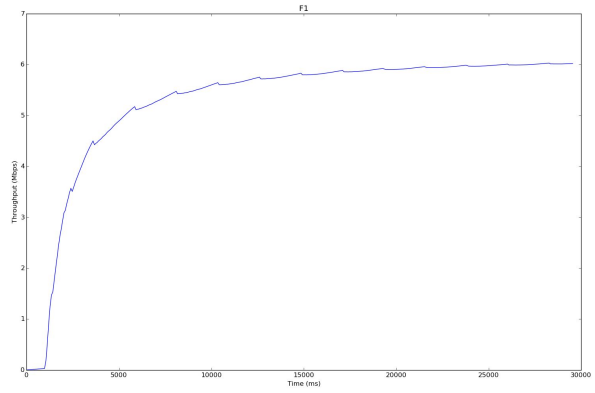
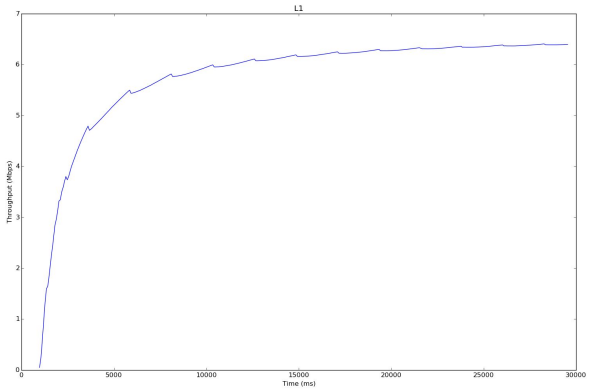
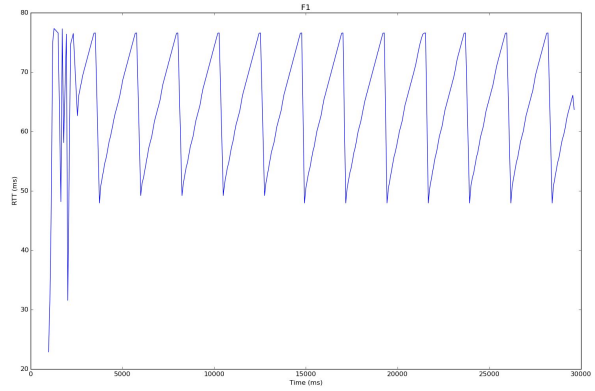
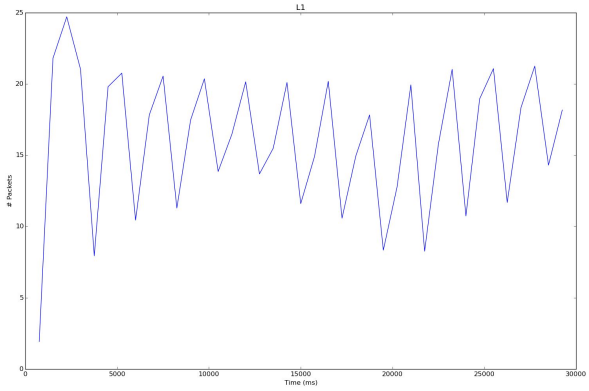
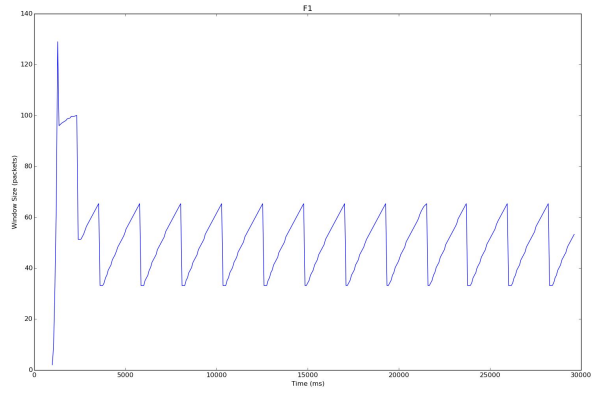
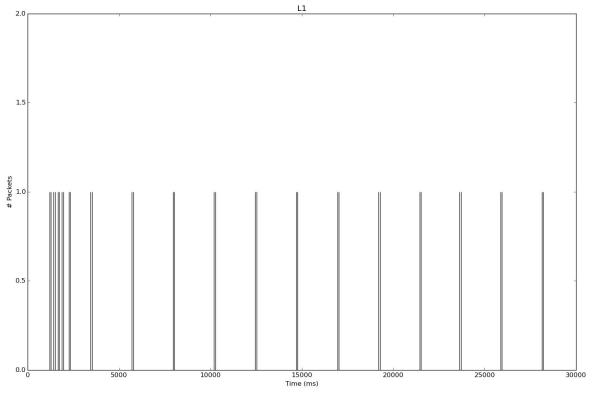
The first thing that was established was basic link functionality for the purpose of test case 0. From there routers were supported just for the purpose of passing packets along with hard-coded tables. A parser for flow specifications was then added. From there, congestion control and dynamic routing were developed separately, with graph functionality being added along the way to provide feedback for debugging.

Results - Test Case 0

FAST TCP

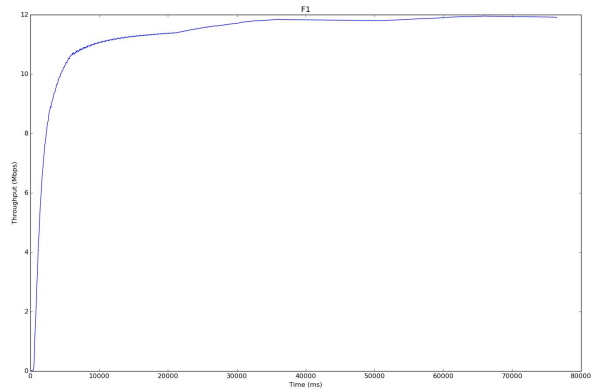
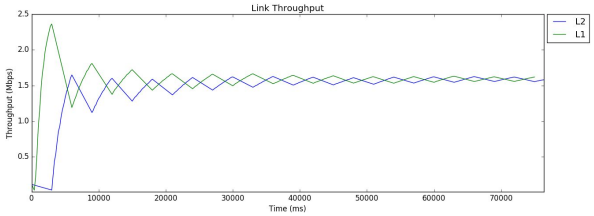
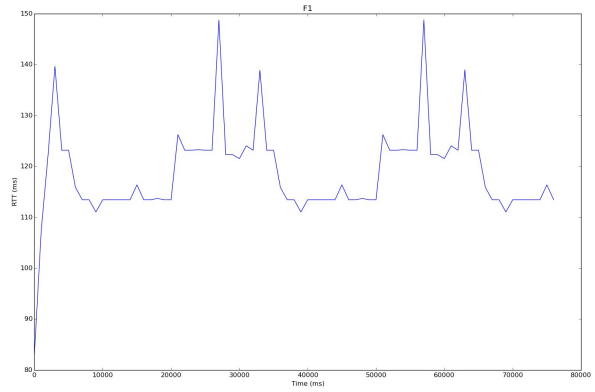
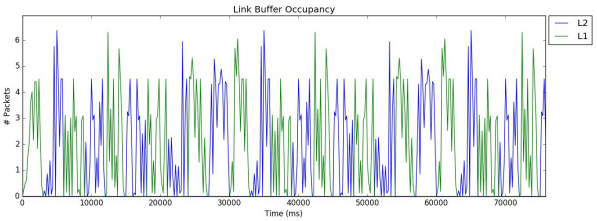
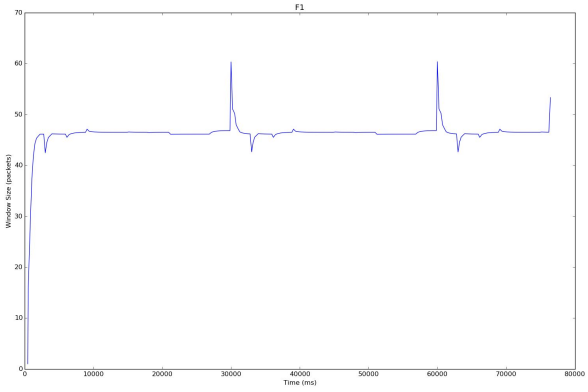
flow throughput	link throughput
	
packet delay	**buffer occupancy**
	
window size	
	

TCP Reno

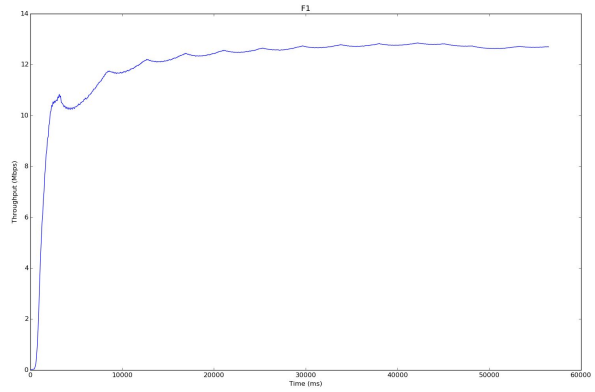
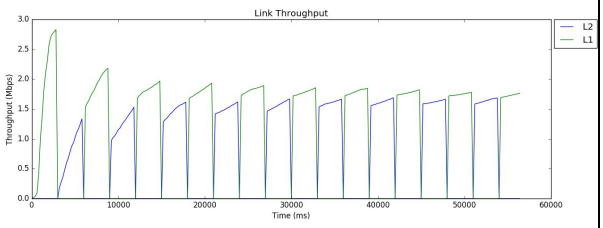
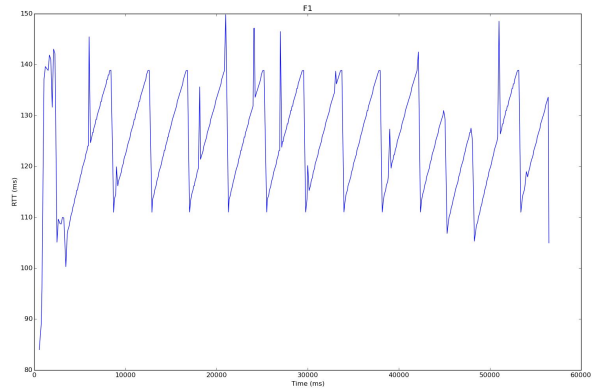
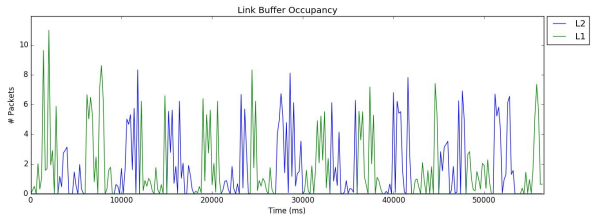
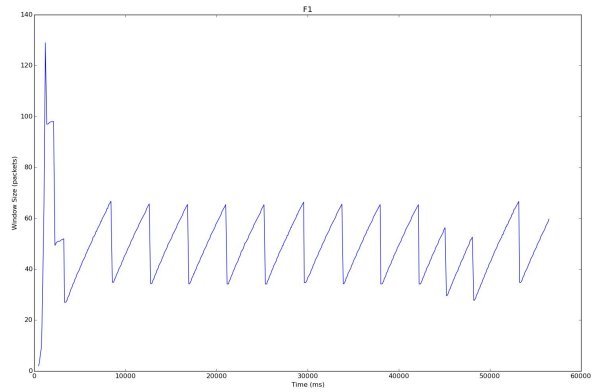
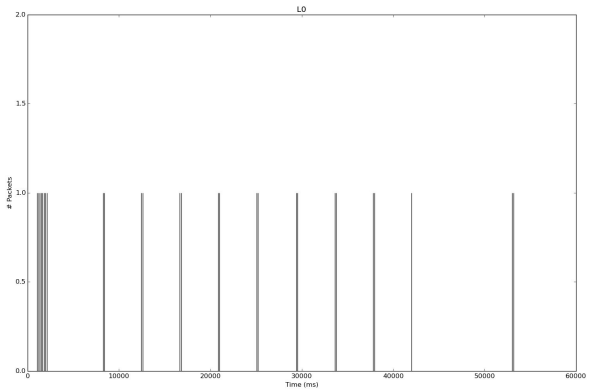
flow throughput	link throughput
	
packet delay	**buffer occupancy**
	
window size	**dropped packets**
	

Results - Test Case 1

FAST TCP

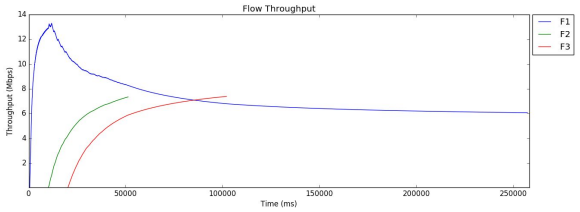
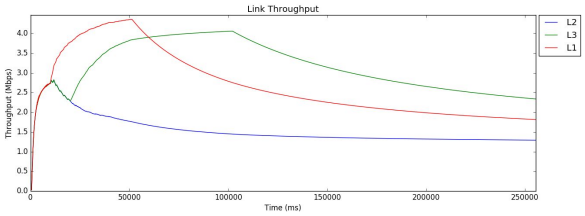
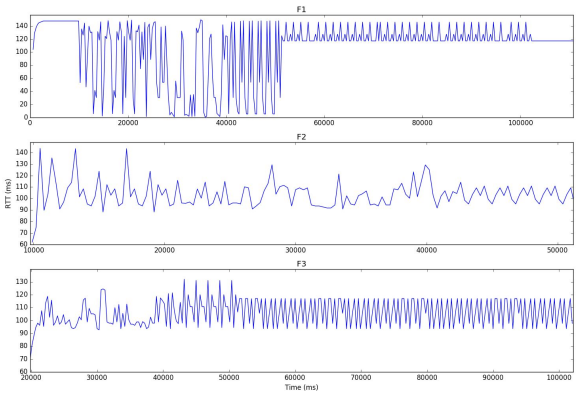
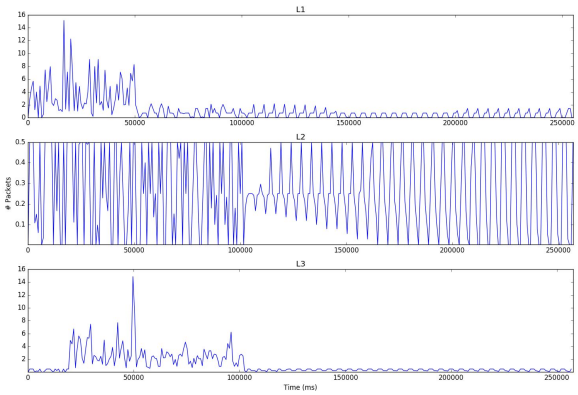
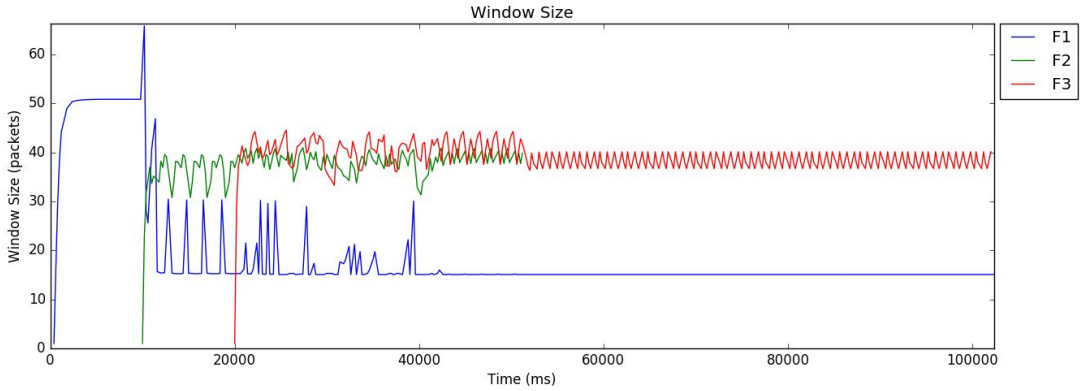
flow throughput	link throughput
	
packet delay	**buffer occupancy**
	
window size	
	

TCP Reno

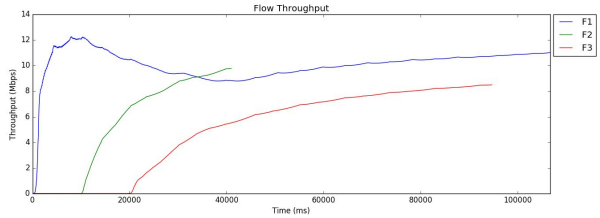
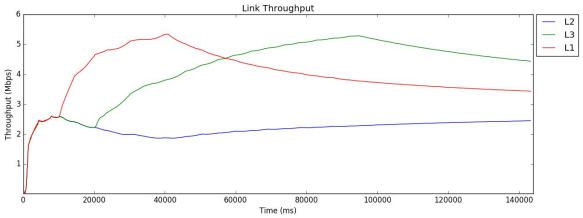
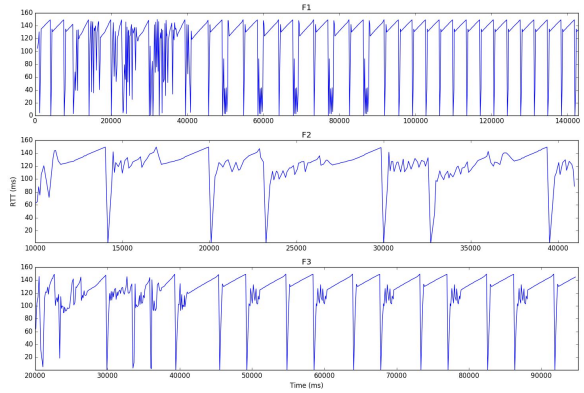
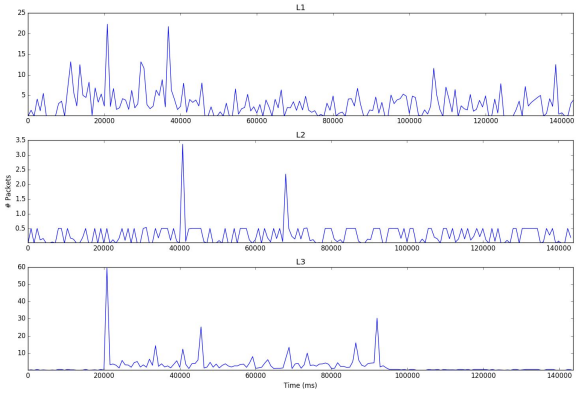
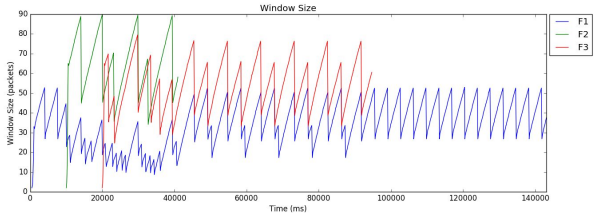
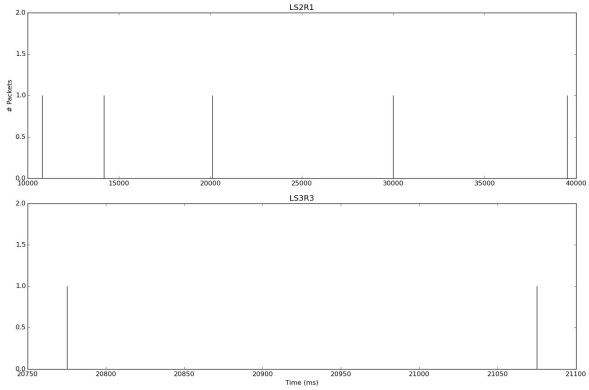
flow throughput	link throughput
	
packet delay	**buffer occupancy**
	
window size	**dropped packets**
	

Results - Test Case 2

FAST TCP

flow throughput	link throughput
	
packet delay	**buffer occupancy**
	
window size	
	

TCP Reno

flow throughput	link throughput
	
packet delay	**buffer occupancy**
	
window size	**dropped packets**
	

Analysis - Test Case 0

FAST TCP

As expected with FAST, the window size reaches a steady state after a second or so of simulation time. The default value for alpha is 15 packets, with update intervals of 200 ms. As we would hope, link throughput smoothly increases as we approach a steady state, and does not decrease. The same applies to flow throughput since there is only one link. The buffer size fluctuates periodically as we would expect, since we are constantly pushing flow packets one way and ACK packets another way and they have different sizes.

TCP Reno

As in FAST, we approach a steady state, but Reno is less smooth in its window size and thus the throughput graphs are less smooth as well. Since Reno handles congestion control by waiting for packet loss, the buffers are generally more erratic and fluctuate between high and low amounts of packets, since we make no effort to control how much is on the buffer. The same applies to flow throughput since there is only one link. At first we drop several packets as our window size grows too large (the first chunk of dropped packets coincides with the first timeout crash in the window size graph). As we approach a steady state we only drop a couple of packets on a periodic basis.

Analysis - Test Case 1

FAST TCP

As in test case 0, using FAST we see an increase in window size up to a smooth steady state; there are occasional drops out of the steady state as a result of dynamic routing. As dynamic routing packets flood the network to create new routing tables, buffers fill up with non-flow packets and thus RTTs for flow packets increase. Thus even though FAST TCP attempts to achieve a relatively constant RTT, dynamic routing causes it to have the erratic behavior observed. Link throughput alternates periodically every 3 seconds between links 1 and 2. The throughput alternates, but eventually approaches a steady state since FAST TCP adjusts the window size to maintain a reasonable steady state throughput in the links. Because of this alternating behavior in link throughput, flow throughput is more erratic than when using static routing alone. Buffer occupancy is as expected since the buffers for links 1 and 2 alternate handling the flow between the two hosts.

TCP Reno

As in test case 0, using reno we have a more jagged window size graph that reaches a large window size during the slow start, then drops and goes into congestion avoidance. This congestion avoidance combined with the alternating behavior caused by dynamic routing leads to very erratic RTTs which still resemble the behavior observed without dynamic routing, but it has occasional spikes because of switches in the routing table delays the transmission of some packets. Link throughput reaches a “steady state” where the throughput changes to the same degree, but it isn’t like the steady state observed in FAST because the throughput of both links do not converge to a common value. The flow throughput is similar to test case 0, but the bumps are larger in size because the changing of the routing tables delay some of the packets. The dropped packets graph is as expected: lots of packets are dropped during the slow start because of the increase in window size, but over time packet drop becomes periodic because the host begins congestion avoidance.

Analysis - Test Case 2

FAST TCP

The jitter in the window size graph we see with FAST is just due to flows not starting at the same time. When the first flow starts, its window size behaves as expected for FAST, but as new flows come in, RTTs rise sharply resulting in heavy oscillations in the window size. This is confirmed by the packet delay graph, which shows erratic packet delays in spaces corresponding to large fluctuations in window size. Similarly, buffer occupancy is very erratic until we reach a steady state in each link because packets are being sent in irregular patterns. Link throughput follows regular FAST patterns, with the curves branching upwards as new flows arise putting more packets on that particular link, until window size regulates the throughput down to a steady state. Flow throughput shows the flows reaching an expected steady state (although flow 2 ends early, hence it is cut off in the graph) over time. Note that in Homework #3, we analyzed an identical network, although the link buffer size was different and the theoretical flows went on infinitely. In theory, flow 1’s throughput drops to a steady state below flow 2 and 3’s, while flows 2 and 3 converge together. While the flow throughput graph does not continue indefinitely, we do see the same behavior in our simulation, as we would expect.

TCP Reno

Reno’s behavior differs from FAST in the same way it did in previous test cases. Flow and link throughput look similar but less smooth, and buffer occupancy and packet

delay vary more wildly. Window size follows the expected periodic pattern with larger variations than in previous test cases just because there is higher traffic.