Network Simulator

Xiangling Kong, Akshay Navada, Jeffrey Sheik, Xiaowen "Soysauce" Zhang, and Zhiyi Huang

Architecture

This network simulator operates a discrete time architecture where events are processed in order and the time is advanced in discrete chunks. An event queue is kept to hold all events experienced by the simulator. The simulation starts by loading in two files, the topology file and the flow file. The topology file (with the file extension .top) first defines hosts and routers using "H" to indicate host and "R" to indicate router followed by the router/host name. Then the links are defined using "L" followed by the link name, source, destination, link rate in megabytes per second, link delay in milliseconds, and link buffer size in kilobytes. The flow file (with the file extension .flow) specifies the different flows for the system.

After these files are loaded into the classes for the links, routers, and hosts, an initial timeout event is created for each flow. On timeout, each flow sends the packets according to the TCP window by creating a send packet event at the time each packet is supposed to be sent at. When these events are triggered, the packets are placed onto the link if the direction is correct and there is space. If not, the packets are placed onto the buffer and if the buffer is full, the packet is dropped. The action of placing a packet onto the link simply means that the link's occupancy is increased and an event is registered for the arrival of the packet at a time in the future equal to the link delay plus the current time. When the arrival event is triggered, the simulator checks if the destination is a host or a router. If it is a host, it generates the ACK packet according to TCP and sends it onto the link again. If it's a router, it will find the link it needs to queue the packet onto through the lookup table at each router. This process repeats until the packet arrives at a host.

We will be using a link state routing algorithm. Each router is initialized with the topology of the network with the delay as the weight. There is another weight called dynamic weight that is initialized to zero. This value will be periodically updated by broadcasting packets to its neighbors. Every few hundred milliseconds, the router updates the link weights it knows and refreshes the time. It then sends a packet of its current routing table to all neighbors. When a neighbor receives this information, it will update all weights in its own table that are newer than what it already has. This ensures that old data isn't getting passed around by the network. After each link update, Dijkstra's algorithm is run on the link to update the routing table. This information is stored as a hash table that maps destination IP address to next router name for ease of use.

Demonstration 1: TCP Reno

To demonstrate that TCP Reno works, a simple test network is created as specified in the example test cases. This test case consists of two hosts connected by a link. A certain amount of information is sent across and data is logged.

The following diagram (**Figure 1**) shows the link rate of the connection over time. As we can see at the very start, TCP begins with slow start where the rate rapidly increases until it starts to drop packets. On packet drop (resulting in triple duplicate acks), the window is halved and TCP enters congestion avoidance where the rate increases linearly until the link starts to drop packets again. This process repeats until all the data have been sent through the channel. In the second diagram (**Figure 2**), the link buffer is show. As we can see, as the link rate increases, the buffer begins to fill more and more until it becomes full at which it begins to drop packets. In the third diagram (**Figure 3**), we see that the window size grows rapidly then grows linearly during congestion avoidance. The size halves whenever a triple duplicate ack is encountered. One interesting thing to note is that during slow start, the window size ramps so high that it experiences two triple duplicate acks due to the amount of packets left in the buffer. Note that these rates are soft averaged in order to smooth it out. In reality, TCP sends packets in

bursts while waiting for acks. This means that with a window size of say 100, in one instance, it may be able to send 100 kb worth of data but then in the next instance there will only be 6.4 kb worth of data in the link due to the acks. Even though the link is empty, the client cannot send any more packets since the acks have not yet been received. This causes the actual graph of the link rate to bounce wildly between 10 mbps and 640 kbps and thus a smoothing algorithm is used (time average).

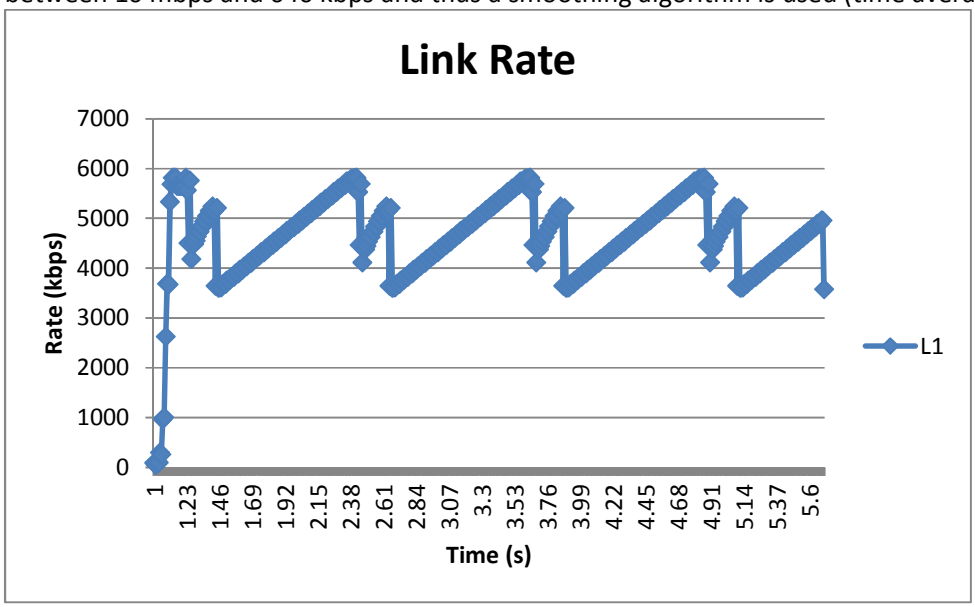

Figure 1. Link Rate of TCP Reno

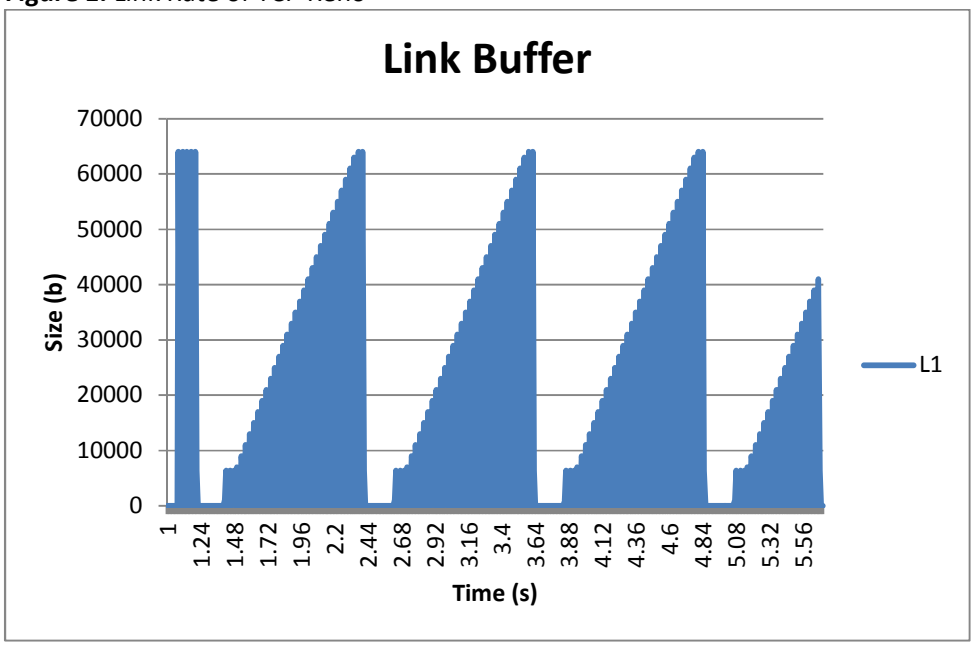

Figure 2. Link Buffer of TCP Reno

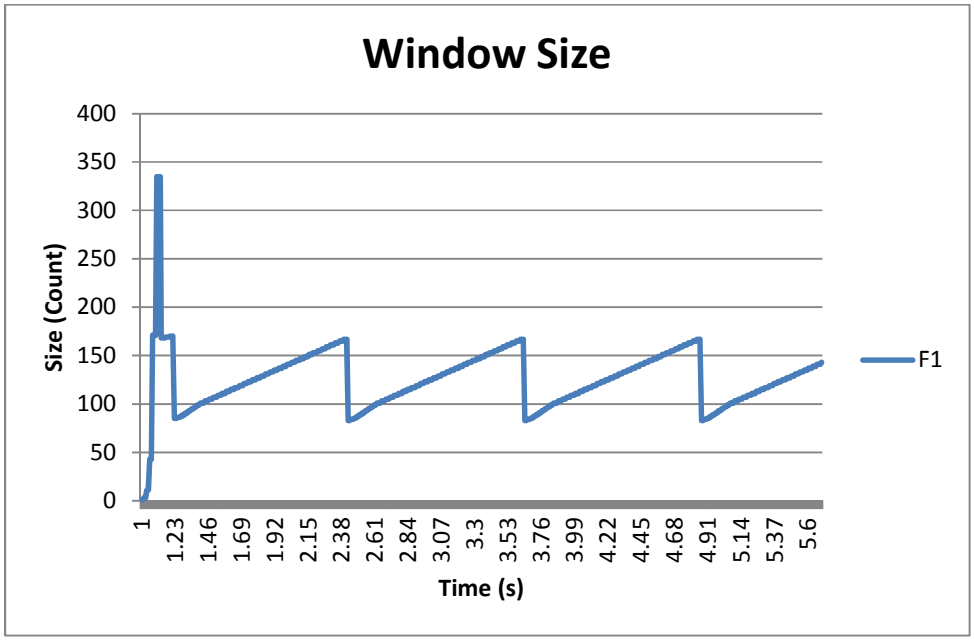

Figure 3. Window Size of TCP Reno

Demonstration 2: Dynamic Routing

In order to demonstrate dynamic routing, the second diamond shaped topology is used from the test cases. In this test case, there are two equal paths to reach the second host. A stream of data is sent between the hosts. In **Figure 4**, the link rate of the two distinct paths are shown. It can be seen that as one link fills up, the routers updates and shares their weights which causes the routing algorithm to put the traffic on the other link instead. As the other link begins to fill up, the routing algorithm switches again. This is why the graph alternates between blue and red (which represents the two links). Note that even though routing is in place, the graph still resembles the standard TCP graph showing congestion control.

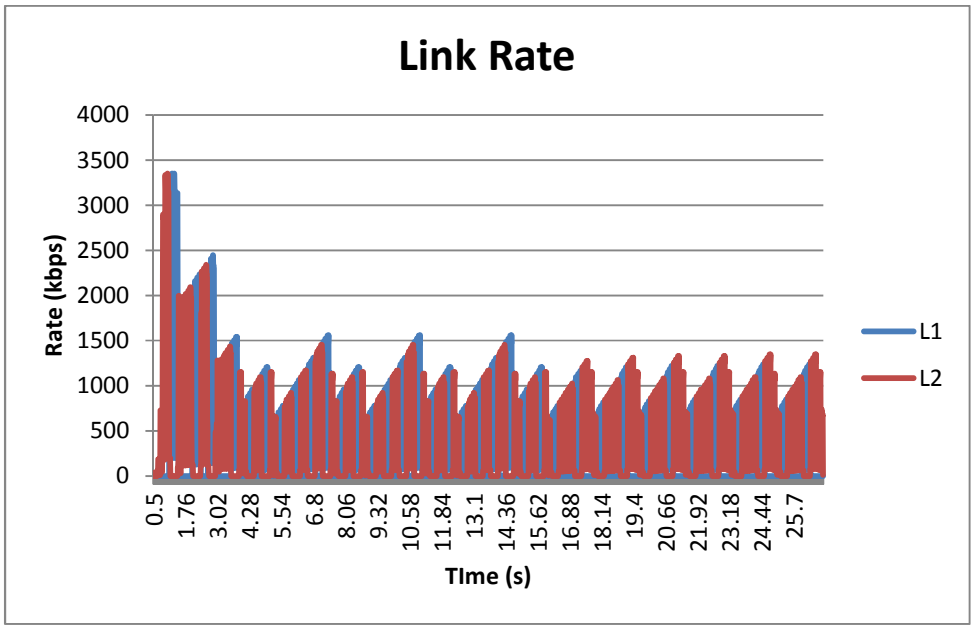

Figure 4. Link Rate of Dynamic Routing

Demonstration 3: Complex Multi Flow

To demonstrate multi flow, the third test case is used (all of these test cases can be found in the data folder). In this test case, there is one main chain of links in which hosts are attached along the way. Three flows between six hosts are then simulated in this environment. In **Figure 5**, which shows the link rates of the three main links (not connecting the router to each host), we can see that it still follows TCP. Over the times of 35 seconds to 64 seconds, the links becomes very congested as all three hosts are attempting to transfer data. When one of the transfers ends, the link rate picks back up.

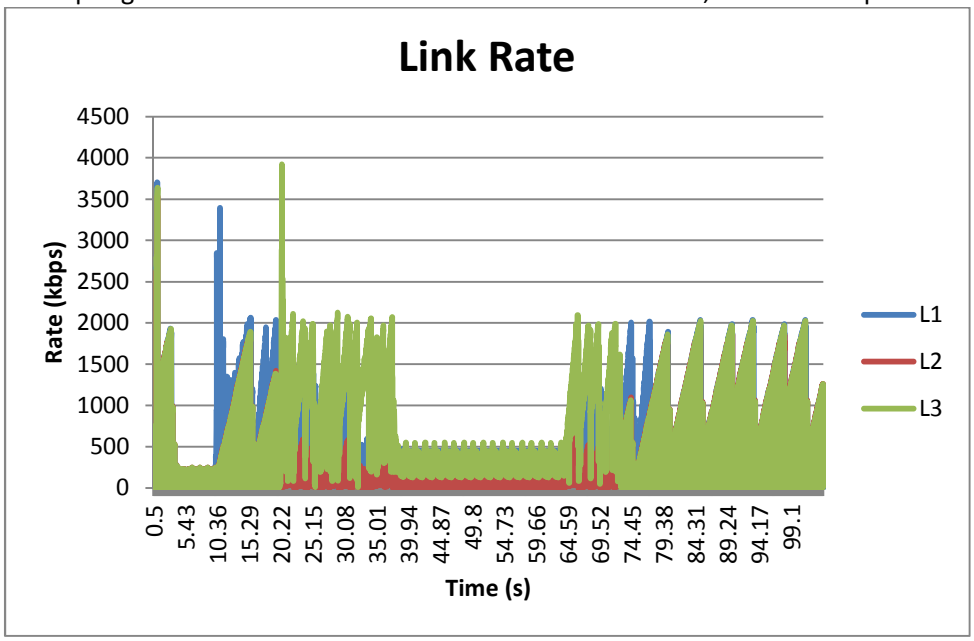

Figure 5. Link Rate of Multi Flow

Purely looking at link rate is not very helpful; instead we can look at how much each individual host sends. **Figure 6** shows the amount of data the hosts S1, S2, and S3 are sending at each instance in time.

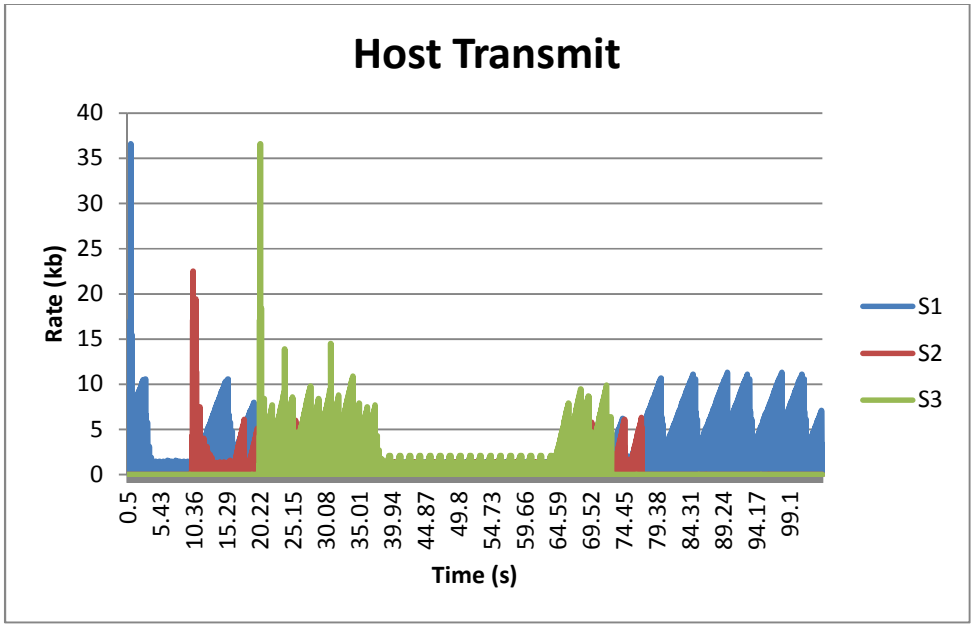

Figure 6. Host Transmit of Multi Flow

We can see that each flow has its own TCP algorithms. However, the twist is that S2 is using TCP Tahoe instead of TCP Reno which features a more costly triple duplicate ack. Instead of halving the window on a triple duplicate ack, it instead resets the window size back down to one. As a result of this, we can see that the red line (S2) is slightly overshadowed by the other two flows (S1 and S3) and also takes a bit longer to finish transmitting everything.

Another interesting graph we can look at is the round trip time. In **Figure 7**, we see that the RTT of each flow quickly stabilizes to what it's supposed to be at with occasional blips on triple duplicate acks (which results from dropped packets).

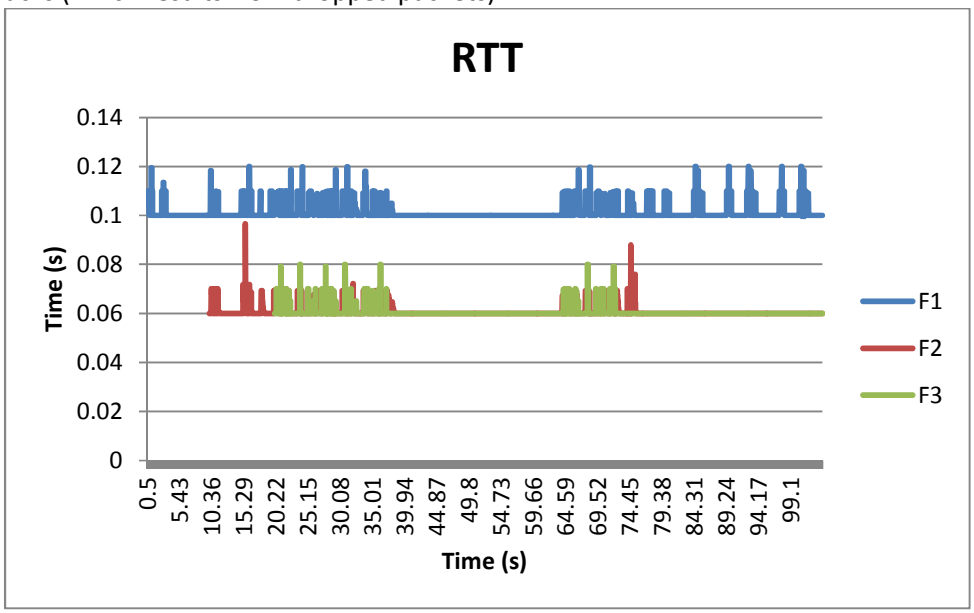

Figure 7. RTT of Multi Flow

How to Use

The application is built in Microsoft Visual Studio Community 2015 which means you need this software in order to compile it. It's free and can be found on the Microsoft website. The executable takes 3 parameters: topology file, flow file, output folder. In the project folder, run.bat specifies an example execution of the executable in the current directory and looks for the input/output files in the Data folder. It dumps the output to report.txt of which an example copy (ran with demonstration 3) can be found in the root folder. The root folder also contains SIM0-SIM2 which is the outputs csv files from each run.