An Abstract Network Simulator

Kangqiao Lei, Eui Han, Jaeryung Song, Eric Nguyen, and Arvind Nagalingam

***Abstract*—This paper details the process and results of generating an abstract network simulator. The simulator allows the user to input any arbitrary network configuration with flows starting at user-defined times. The program simulates the information transfer through the network and generates an output file. The output file can be put into the companion Matlab program to generate graphs of network performance. The simulator uses TCP protocol to transfer packets and the Bellman-Ford algorithm for routing.**

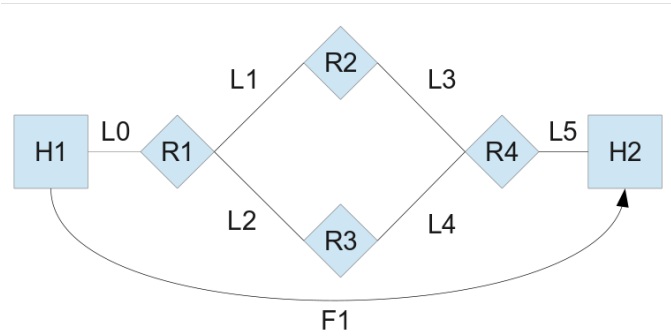
***Keywords*—Internet, Hosts, Routers, TCP, Bellman-Ford**

I. Introduction

The Internet is a complex system of interconnected clients that communicate based on international network protocols. The internet is a network of networks which connects millions of local networks like schools, businesses, and government offices to all other networks. Each individual local network can look very different, but they are all essentially built on three basic network objects: hosts, routers, and links. The number of hosts and routers, as well as the links that connect them to each other, can vary dramatically among all possible networks. Furthermore, different algorithms can be implemented for routers to efficiently transfer packets from host to host in the network. And in addition to this, different internet protocols for communication can be implemented to manage congestion in the network and guarantee all information is transferred.

In this paper, the team of authors designed an abstract network simulator that allows the user to generate an arbitrary network of hosts and routers. After this, the user can define flows of any size that can begin sending at any predefined time using either TCP Tahoe or TCP Reno. The program will then simulate the information transfer over discrete time steps. When the simulation is finished, an output file is generated which lists all of the occurrences of important events in the network. This output file can then be inputted into the corresponding Matlab program for our simulator to generate graphs of the performance of the user-created network over time.

The paper continues by detailing how the network simulator was implemented in C++ and is structured as follows. In Section II, an overview of how the network simulator generates the network and simulates events is described. In Section III, a description of the TCP protocol implemented in our network is given. In Section IV, the implementation and advantages of the Bellman-Ford algorithm for routing efficiency are shown. In Section V, the results of our network simulator for the three test cases are portrayed. And finally in Section VI, the paper is summarized and opportunities for improvement in the design are explored.



II. Network Simulator Implementation

Our network simulator implements object oriented design and priority queue approach to handle events. First there are classes such as Host, Router, Link, and Flow, to set up a network topology. Because they are designed as general objects, there is no restriction in the number of objects and how they are connected in the network. Then there are Event classes that handle different kinds of events such as sending and receiving packets, ACK timeout, etc.

These are steps of how our simulator works: First it takes input from .JSON file. JSON file must include a full specification of the network, including how many nodes there are, their ID's, link buffer, delay, flow size, etc. Then we parse the input using c++ rapidjson package (Free under MIT license) and create a network object. We start the simulation by setting up the flow and routers: Flows are divided into packets and routers register so that they know how many nodes each of them are connected to. At this point, a router only knows the data rate of links it is directly connected to. Then we trigger the whole simulation by sending the first packet.

The host pushes a SendPacket event into the event queue. Event queue pops the SendPacket event and tells the connected link to receive the packet. If link is busy (already transmitting another packet), it pushes the packet into its buffer. If its buffer is full, it just discards the data. As it pushes its packet to the link, the host also registers another event called ACK Timeout event. When this event is popped later at its scheduled time, the host checks if it has received the ACK packet of corresponding number. If it did, the event is thrown away and does nothing. If it didn't, the host resends the packet. (Selective Repeat)

When a router receives the packet transmitted through a link, it checks the final destination of the packet and picks the best route to the destination. (Bellman-Ford) But at first, the router does not have a complete routing table. So whenever it checks the routing table and the next hop is undefined, it uses the Greedy Algorithm instead. It checks every link it is connected to, and chooses the one with minimal occupancy and maximum

data rate. When using the greedy algorithm, some checks are done to ensure that packet does not go back to its source. Every 20 seconds, the router sends out the control packet and informs its neighbors about its change in the routing table. Ideally this can be done whenever there is a change in the routing table. But since it can crowd the link very fast, we decided to send out the routing packet every 20 seconds.

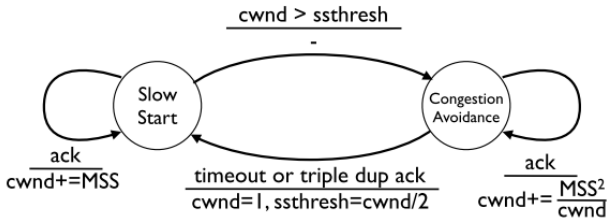
When a packet arrived at its destination, the host stores it and responds to its source with an ACK packet. ACK packet number is decided by the highest number in a sequence without a gap. For example, if the received packets are 4,5,6,9, the ACK number will be 6. Then the ACK packet travels back to the source. When a host receives an ACK packet, it records its sequence number and check if it is a duplicate ACK. At third time, the host concludes that there is a congestion in the network and resends the data packet. The event continues until every flow has ended or the maximum simulation time (defined in a global header file) is reached.

III. TCP Implementation

Transmission Control Protocol (TCP) is one of the two main transport layer protocols. TCP guarantees that the data packets beings transported from host to host will eventually arrive without error and in the order they were sent. These attributes are the reasons why TCP is considered a reliable service. There are many variations of TCP that exist. These variations address how time efficient the protocol is at delivering the same size payload. We choose to implement TCP Tahoe which uses a very simple version of TCP that has very basic congestion control.

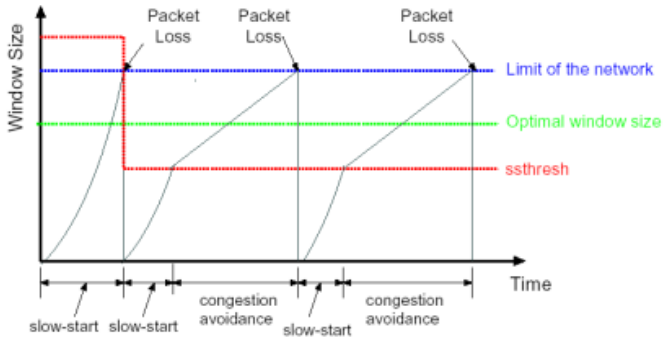
There are some underlying assumptions made in order to simplify the protocol. First, TCP is connection oriented such that the two end hosts must hand-shake for connection establishment and maintenance. The three-way-handshake involved in forming a connection is modeled by delaying the start of the flow. Second, the receive buffer size is assumed to be for flow control. This means that the application is accepting the data in the packets no matter how fast they are flowing into the destination host. This means that TCP only needs to examine congestion control which is limited by the link buffers. Lastly, the links are modeled as a buffer queue. Also, links never cause bit errors. Instead, packets are only dropped if the link buffer is full. Thus, if there is ever a triple duplicate acknowledgement (ACK) in the connection, it will be due to congestion instead of a packet lost to bit-errors. Thus, a triple duplicate ACK should be treated as if there were a time out. This is another reason for using TCP Tahoe.

Our implementation of TCP can be broken down into 2 components. The first will be reliable data transfer with congestion control. The second will be discussing timeout. The finite state machine (FSM) for our implementation of TCP Tahoe is shown below:



As seen in the FSM, there are two modes of operation for TCP: Slow Start and Congestion Avoidance. The congestion window (cwnd) determines how many packets are allowed to be in the network. In Slow Start, cwnd grows exponentially by incrementing cwnd for every ACK received. This doubled cwnd for every round trip travel time (RTT). For example, if 4 ACKS were received sequentially at the sending host, 8 data packets would be sent since cwnd would increment from 4 to 8. In Congestion Avoidance, cwnd grows by one for every RTT.

The TCP connection transitions from Slow Start to Congestion Avoidance only when cwnd exceeds the slow start threshold ssthresh. The connection transitions from Congestion Avoidance to Slow Start only when there is a time out or triple duplicate ACK. When this occurs, cwnd will be set at 1 and ssthresh will be set to half of the current cwnd. This means that the connection will enter congestion avoidance earlier. Ideally, cwnd will approach a steady state where the cwnd will be as optimal as often as possible as indicated by the following graph.



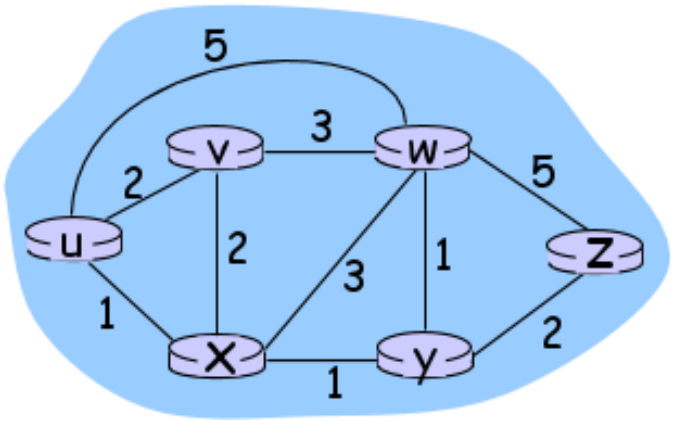
Timeout is calculated by timing packets from the time they are sent to the time they are acknowledged. Only one packet is timed at any given moment and the time packet must contain a sequence number that has not already been sent. This sample of the round trip travel time is denoted as sampRTT. Once a sampRTT has been computed, a running estimate on the RTT is computed by the following: estRTT = 0.875*(estRTT) + 0.125*(sampRTT). Next, a estRTT deviation is calculated by: devRTT = 0.75*(devRTT) + 0.25*(—sampRTT - estRTT—). Finally, the optimal timeout value is computed as: TO = estRTT +4*(devRTT) + 0.1. The constant 0.1 at the end was used to make sure timeout never converged to exactly the estimated RTT such that devRTT was 0. This ensures every

ACK has an extra 0.1 seconds to reach the sending host before the timeout. The coefficients of the three update equations were the values recommended by [RFC 2988].

This simple version of TCP guarantees that all the packets will be eventually received at the destination host in order. However, without fast retransmission and the harsh cwnd reset to 1, the speed of this implementation will be slightly slower when compared to TCP Reno which uses fast recovery.

IV. Routing Algorithm Implementation

The Bellman-Ford algorithm was implemented in our network in order for the routers to dynamically choose the optimal path from host to host. Bellman-Ford is a distance vector based algorithm that attempts to find the optimal next hop router/host for a packet. The Bellman-Ford algorithm is based off of each router storing the cost to send an incoming packet to all of its neighbors as well as the distance that each of its neighbors is from the destination host.



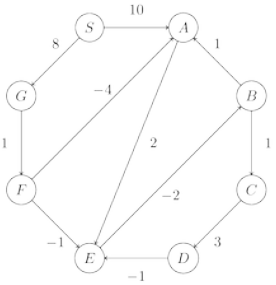
Bellman-Ford quickly finds the least cost path to the desired host in order to efficiently move packets through the network. It is also more memory efficient than Djikstras algorithm since it finds the least cost path by only storing the distance vectors of ones neighbors and the costs to those neighbors, which is a lot less memory than storing the distance vectors of every router in the network to every other router in the network. Furthermore, less control packets must be sent in the Bellman-Ford algorithm than Djikstras algorithm since less distance vector data needs to be sent among routers and not all routers need to know all distance vector data in the network.

The first step of the Bellman-Ford algorithm is to get the costs of sending a packet to its neighbors. This is accomplished in our network by examining a metric that we call the link cost which is stored in the links. Link cost is equal to the propagation time of a packet through the link plus the occupancy of the buffer divided by the link rate, or throughput of the link; in equation form, LinkCost = PropagationTime + (BufferOccupancy/LinkRate). This gives an estimated time for the packet to be transmitted from the current router to its neighbors, which can be a mix of hosts and routers. Each link cost is tabulated in a Cost Vector which the router updates every timestep of the simulation.

The second step of the Bellman-Ford algorithm is to get the distances and next hops of neighboring nodes from the destination host. This is accomplished by sending control packets throughout the network. Each router will equate the link cost of a host that it is one hop away from as the distance to that host. However, routers do not initially know the distances to all other hosts that it is not directly connected to. This information must be propagated through the network using control packets. Until these distances are known through control packet communication among routers, the router will assume that the optimal path to the destination host is the neighboring router with the least cost.

On every timestep, each router will add its cost vector to get to each of its neighbors to the distances that each of its neighbors are away from all other hosts. Then the router will take the minimum distance+cost to each host and store it as its own distance to the each host. It will also call that router which it copied the minimum distance+cost from as the next hop for each host. This way, the optimal path to every host can be found with little memory storage. Bellman-Ford is therefore a very effective distance vector routing strategy for larger networks with multiple routers and hosts, like the network presented in Test Case 2.

To encompass all aspects of the Bellman-Ford algorithm in our network, each router object has a map called the distance vector table that stores individual distance vectors for each neighbor. The distance vector stores a pair for each host in the network. The first entry of the pair is the distance to that host. The second entry of the pair is the next hop node to that host. Furthermore, each router object also has a vector call the Cost vector that stores the link costs to all of its neighbors. Finally, control packets are generated every 20 seconds to allow routers to update other routers on changes to distance vectors in their own table. The control packets are restricted to every 20 seconds in order to prevent network congestion with control packets. 20 seconds was chosen experimentally after seeing how frequently the network needed to be updated in order to ensure that packets were being sent along the least cost path while keeping control packets at a minimum.

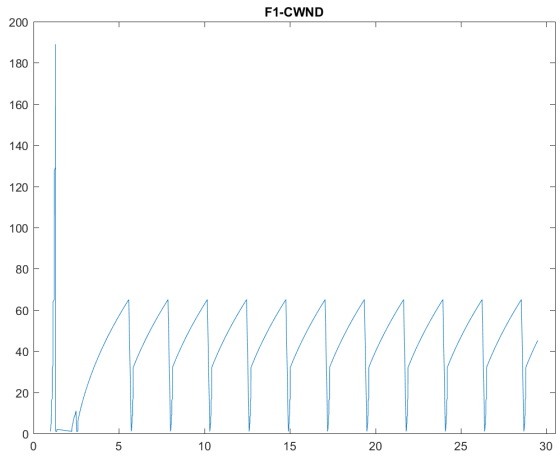


	Iteration							
Node	0	1	2	3	4	5	6	7
S	0	0	0	0	0	0	0	0
A	∞	10	10	5	5	5	5	5
B	∞	∞	∞	10	6	5	5	5
C	∞	∞	∞	∞	11	7	6	6
D	∞	∞	∞	∞	∞	14	10	9
E	∞	∞	12	8	7	7	7	7
F	∞	∞	9	9	9	9	9	9
G	∞	8	8	8	8	8	8	8

V. Results

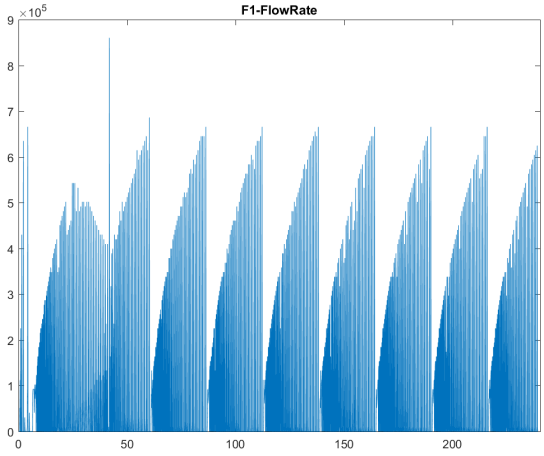
All graphical results are provided in the plots folder of the code submission. Selected plots are shown on the next page.

Test Case 0:



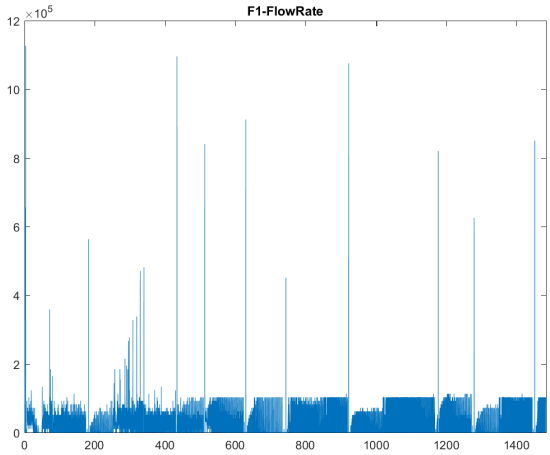
The easiest graph to look at is CWND for the flow. The flow transmits as much as possible and times out. Upon the return of the packet timed out, the congestion window is allowed to grow until it times out. From there it enters slow start (exponential growth) and then congestion avoidance (linear). The flow rate, Packet delay, and Buffer occupancy follow the CWND graph closely. The drops are due to the return trips of the acks. The spike of packets lost in the beginning sees a large loss and then packets are slowly dropped every time the CWND peaks and times out.

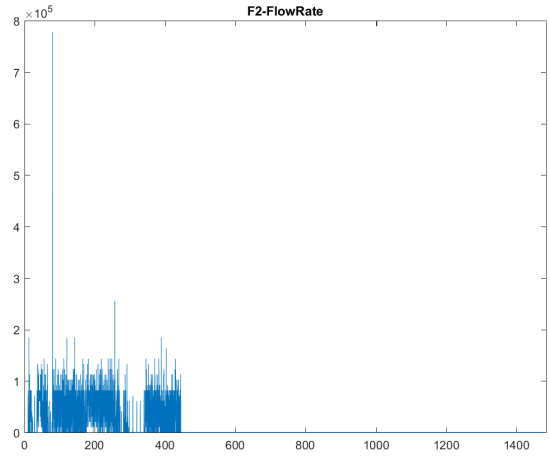
Test Case 1:



The purpose of this test case is to test the routing algorithms. The flow CWND follows a very similar shape to test case 0. The entry and exit links L0 and L5 show the buffer occupancy being filled. L5 receives data payloads in contiguous streams due to smart routing and due to the instantaneous reporting, the graph doesnt look like a block. Our routing algorithm primarily uses L2 and switches to L1 utilizing both branches.

Test Case 2:





The flow rate graphs show intuitive results. Flow 1 initially starts and the flow rate is cut down when the other flows enter. Flow 2 finishes quickly since it has the smallest payload and smallest distance. Flow 3 has to compete with Flow 1 when it transmits.

VI. Conclusion

Implementation of the abstract network simulator was successful. In all test cases, all flows were efficiently transmitted in a timely manner. Furthermore, the graphs that were produced for all test cases were consistent with our expectations of the proper functioning of the network.

Though our network simulator did work as we intended it too, there are many ways which it could be improved in the future. One example of this could be to allow for hosts and routers to join and leave the network while the simulation

is running. This would more accurately represent certain networks, in particular a basic wireless network. Another potential test case could be for multiple hosts to be connected to one first hop router. Our network simulator would most likely be able to handle this kind of network, but it was not particularly tested. Finally, a more user-friendly network generation mechanism could make using the simulator easier. Currently the user must input a text file that has the correct syntax for the simulator to recognize it, and it is not visually apparent what the network looks like. A GUI could be created that is easier to interact with the simulator.