CS 143: Final Report

Rafael Fueyo-Gomez Cortland Perry Kelsi Riley Sakthi Vetrivel

December 10, 2018

1 Introduction

For our term project, our team decided to pursue the network simulator. At a high level, the simulator takes in a input of a the description of network, such as its hosts, routers, links, and flows, runs a simulation for a user-specified duration, and outputs graph demonstrating the progression of certain metrics, such as flow rate, window size, or buffer occupancy, after the end of the run.

This simulator is written in Python3 and makes use of the matplotlib package. Code can be found at `https://github.com/rafugo/cs143_simulator`. Instructions to run the test cases can be found in the repository's README file.

2 Simulator Architecture

2.1 Simulator Class

Our simulator class contains all of the network objects needed to simulate our network.

- **__init__(self, string filename)** This function uses the input file specified in filename to create network objects based on a network description.
- **plot_metrics(self)** This function uses all of the metrics that were recorded during simulation and plots them. The graphs are then outputted as png files in the directory of the code.
- **run(self)** This function runs the actual simulator. Upon starting the simulator, we first have all of router make their handshakes to learn the routing table for the network. Then, we run all network events on a timestep basis. So, for a user-specified amount of time, we run the simulator for so many timesteps. For every timestep, we make calls to link class to try to send all packets in the buffer, and calls to every flow to send packets in the flow. We then increment our global clock. Moreover, every five seconds, we send all of our link states throughout the router to update the link states and the routing table if needed.

2.2 Host Class

Our host class simulates the behavior of a host in our network.

- **__init__(self, int hostid, int linkid)** Initializes a host object. The hostid allows us to identify against different host objects and the linkid allows to identify which link is associated with this host.
- **send_packet(self, Packet p)** Send the packet p by adding (or attempting to add) the packet to the link buffer.

- **receive_packet(self, Packet p, int linkid)** Sends acknowledgements of packets received and notifies the correct flow of these acknowledgements.

2.3 Router Class

- **__init__(self, int id, List¡Link¿ links)** Initializes a router object. The id allows us to identify different routers, and the list of links refers to the links that are connected to this router object.
- **receive_packet(self, Packet packet, int linkid)** Processes packets and has case handling for the different packets it might receive such as handshake packets, handshake acknowledgements, standard packets or routing packets.
- **forward_packet(self, Packet packet)** Forwards packets based on the routing table and their destination.
- **send_handshake(self)** Send the initial handshake packet to the adjacent routers to determine which nodes are connected.
- **recalc_link_state(self)** Recalculates the link states.
- **send_link_state(self)** Send out our link state array to neighbors.
- **receive_handshake_ack(self, Packet packet, int linkid)** Processes handshake acknowledgement.
- **receive_link_state(self, string array state_array_actual)** Upon receiving a link state, the router updates its own link state array.
- **run_dijkstra(self)** Run's Dijkstra's algorithm to determine the routing table.

2.4 Packet Class

- **__init__(self, int sourceid, int flowid, int destinationid, int packetid, string packet_type, string data = ")** Initialize a packet object. sourceid referes to the id of the source (whether it's a router or a host). flowid refers to the id of the flow the packet belongs to. destinationid referes to the id of the destination (whether it's a router or a host). packetid is the identification of the packet within the flow and is its number in the sequence. packet_type is the type of the packet and can take on the values of STANDARDPACKET: a normal packet, ACKPACKET: an acknowledgement packet, ROUTINGPACKET: a routing table packet, SYNPACKET: a synchronization packet, SYNACK: a synchronization packet acknowledgement.

2.5 Link Class

- **__init__(self, int linkid, int connection1, int connection2, float rate, float delay, int buffersize, bool track1=** Initializes a link object for our network. linkid is the ID for this link object. connection1 and connection2 are the id's of the hosts/routers at the ends of the link. rate and delay are floats that refer to the rate and propagation delay of the link. buffersize is the size of the buffer for this link. track1 and track2 are boolean values to determine if we should track metrics for this link.
- **add_to_buffer(self, Packet packet, int sender)** Add packet to buffer based on the sender of the packet.
- **send_packet(self)** Tries to send a packet on both of the half links corresponding to the link.
- **HalfLink** Represents one direction of the Link. (i.e. all packets travelling across a given HalfLink will be going to the same destination) Class has its own definitions for add_to_buffer and send_packet.

3 Congestion Control

Our congestion control takes place in our flow class. For our project, we've implemented two congestion control algorithms, TCP Reno and TCP Fast. To do this, we have two different Flow classes, with different implementations of the same public methods. Here's an overview of all of the public methods.

- **run(self)** This function is called at every timestep by the simulator for the flow to process any acknowledgements or send any packets as necessary for that timestep.
- **process_ack(self, Packet p)** Process an acknowledgement once it's received by a host. Checks for duplicate acknowledgements and changes congestion control states, window size, and threshold accordingly.
- **send_packets(self)** Send the packets for the flow depending on the current system time and the acknowledgements that have been received thus far.

4 Metrics Tracking

5 Results

5.1 Test Case 0

5.2 Test Case 1

5.3 Test Case 2

6 Timeline & Division of Labor

6.1 Timeline

- **Weeks 4-6:** Completed initial architecture design of simulator and initial implementation of a few simulator classes, such as Packet and Link classes
- **Weeks 7-8:** Completed initial implementation of all class in the simulator, started debugging Router and Flow classes
- **Week 9:** Finished debugging fundamental architecture for simulator, started work on implementing and debugging TCP Reno congestion control
- **Week 10:** Finished implementation of TCP Reno Congestion Control, debugging metric tracking
- **Week 11:** Finished implementation of TCP Fast, finished report and presentation

6.2 Division of Labor

- **Rafael Fueyo-Gomez:** Initial architecture design, routers, flow, congestion control (TCP Reno and TCP Fast)
- **Cortland Perry:** Initial architecture design, routers, congestion control (TCP Fast)
- **Kelsi Riley:** Initial architecture design, initial implementation, metric tracking, congestion control (TCP Fast)
- **Sakthi Vetrivel:** Initial architecture design, flows, congestion control (TCP Reno), report & presentation slides