Ceng 435
Term Project - Part 1 Report
Report-Group 14

Hilal Ünal

2172112

Eda Özyılmaz

2171882

I. Introduction

This document is a report for Ceng 435 Data Communications and Networking Term Project part 1, we developed UDP socket application for this homework. We simply described our design and implementation approach, our methodology, and our experimental results with their explanations and our comments to them.

We calculated the link cost of given topology and determined the shortest path by using Dijkstra's shortest path algorithm. Also, as stated in the homework pdf, we completed the experiments on our project, and plot a graph which shows emulated delay versus end-to-end delay. We submitted our work as "discoveryScripts" folder which contains scripts for calculating link costs and "experimentScripts" folder which calculates end-to-end delay and represents shortest path from s to d. The detailed explanations about our project can be found in the following sections.

II. Explanations

This part contains our explanations, methods, and our comments on homework's parts. The homework explained in basic parts, which are A.About Topology, B.Link Costs, C.Finding the Shortest Path using the Dijkstra Algorithm, D.Experiment, and E.Workshare. This report also includes Dijkstra's shortest path algorithm tables and a figure that provides the relation of network emulation delay and the end-to-end delay.

A. About Topology

After downloaded download_topology.xml, we created slice from this xml by using Geni platform. After creating the slice with the xml file, we used the drop-down list on Ste 1, and selected a GENI to reserve our resources at. For these modules, InstaGENI aggregates are recommend; therefore, we used one of the InstaGENIs which was NPS InstaGENI. Then we click the "Reserve Resources" button and created our topology. After reserving resources, we went back to the slice and wait for our resources to be ready, when they are ready they turn into green. Then we could obtain IP's of links and port's for the nodes. To run the ssh we used "e2172112@pc2.instageni.nps.edu" as IP and different port numbers for each node, this port number are obtained from topology. We used the following ssh command lines to connect to the nodes d, r1, r2, r3, and s of this topology:

$\$\ ssh\ -i\ /.ssh/id\_geni\_ssh\_rsa\ e2172112@pc2.instageni.nps.edu\ -p\ 26610$

$\$\ ssh\ -i\ /.ssh/id\_geni\_ssh\_rsa\ e2172112@pc2.instageni.nps.edu\ -p\ 26611$

$\$\ ssh\ -i\ /.ssh/id\_geni\_ssh\_rsa\ e2172112@pc2.instageni.nps.edu\ -p\ 26612$

$\$\ ssh\ -i\ /.ssh/id\_geni\_ssh\_rsa\ e2172112@pc2.instageni.nps.edu\ -p\ 26613$

$\$\ ssh\ -i\ /.ssh/id\_geni\_ssh\_rsa\ e2172112@pc2.instageni.nps.edu\ -p\ 26614$

We used these information to determining link costs and for creating scripts. We obtained the following topology with the given xml file for our project, Figure 1:

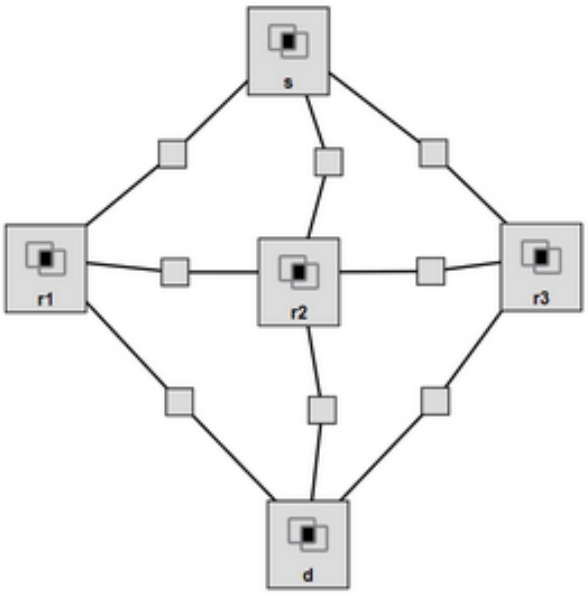

Fig. 1. Given topology

This topology contains five nodes as s, r1, r2, r3, and d. We developed UDP socket application. UDP, which is

User Datagram Protocol, is used for simple request response communication. Before we start, we needed to run the shell scripts "configureR1.sh" and "configureR2.sh" on only the node r1 and r2. We have one script for each node as s.py, r1.py, r2.py, r3.py, and d.py and we only execute them once. The detailed explanation for executing the scripts are given in the README file.

According to our design, source nodes read message line by line from the "deneme.txt" file which has 1000 lines and it is in the same directory with script files. We used python to implement our code. Every node which is ready to receive some message sends message that they are ready to the source node. After source node receives this ready message, it encodes every line and send it to the receiving node. We first calculated the link costs and found to shortest path by using Dijkstra's algorithm.

B. Link Costs

We find the costs of all links with RTT, which is round-trip time, as the cost metric.To calculate link costs, we used the scenario which explained below:

- The s node is server for r1-r2-r3
- The r1 node is client for s-r2-d
- The r2 node is server for r1-r3 and client for s-d
- The r3 node is client for r2-s-d
- The d node is server for r1-r2-r3 nodes

When a node is server, it reads the "deneme.txt" file and sends the lines one by one to the destination nodes. If a node is client, it sends message which states it is ready to receive message to the source node. After source node receives ready message, it send the line that is read from the "deneme.txt" file. This basically explains how or code works. The more detailed explanation can be found on s.py, r1.py, r2.py, r3.py, and d.py file with detailed comments on them. We calculated link costs and stored them on the nodes as stated in below:

- r1 stores the link costs for (s-r1, r2-r1, d-r1)
- r2 stores the link costs for (s-r2, d-r2)
- r3 stores the link costs for (s-r3, r2-r3, d-r3)

To find the link cost between nodes, we first created array name linkCosts[] and then we calculated the time between the start of *msg, addr = server.recvfrom(2048)* and the time at the end of this line, and subtract them with each other. To get time, we used *start = time.time()* and *end = time.time()* lines. Then we divided the final result by 1000 because our txt file contains 1000 lines, so we obtains the average link cost. This gives the time passed for receiving message from the source node. Only r1, r2, and r3 nodes stores the link cost. They each generates output file named "linkcost_rx.txt" which gives the stored link costs in seconds. This file is generated in the same directory as "rx.py" file. The output file has lines like "Link cost r1 to s: 0.121705". The number of lines in the "linkcost_rx.txt" file is the same as the number of links node rx has.

While running the script we created we got the following error: [Error no 48] Address already in use error. After making research about this error, we figured out that this error means there is a process running on the given port, a socket is left open. To overcome this error, we used the following command lines:

$ ps -fA || grep python
$ kill [processNo]

With these command lines, we found which process is left open, and kill that process to use the necessary addresses.

According to the homework pdf, each node has to be able to send and receive messages at the same time ,multiple requests, from its neighbors. Therefore, we used threads to accomplish this feature. Each node generates necessary number of threads to deal with every link at the same time. For example, r1 has three links as s-r1, r2-r1, and d-r1, so it generates three thread. With this method, each link can send and receive messages at the same time. First we forgot to add the *thread_t1.join()* statements on our code. Therefore, when we tried to generate "linkcost_rx.txt" after generating and starting all of the threads, we couldn't get the exact link costs, we obtained all link costs as 0. Because we were assigned 0 to all of them on global. After we joined all of the threads by using *thread_all.join()* method, we obtained correct link cost results from the linkCosts[] array.

C. Finding the Shortest Path using the Dijkstra Algorithm

In the homework pdf, it is stated that we should find the shortest path from node **s** to node **d** by using Dijkstra's shorting path algorithm. However, implementation of the algorithm was not necessary so we calculated the shortest path manually. The table which represents source node, destination node, and link cost as milliseconds between them is constructed can be found below, as Table 1: Link Cost Table :

TABLE I
LINK COST TABLE

Source	Destination	Link Cost
s	r1	0.121705
s	r2	0.161641
s	r3	0.000917
r1	r2	0.140900
r1	d	0.060932
r3	r2	0.080814
r3	d	0.000536
r2	d	0.080904

After finding all, link costs we applied Dijkstra's shortest path algorithm. As stated in homework we try to send message from node s to node d in shortest path possible. According to this algorithm, we first calculate the link costs from s, then we go to the node with the smallest link cost and recalculate the link costs.

In our example, when we started from node s, link cost of s-r1 was 0.121705, s-r2 was 0.161641, s-r3 was 0.000917. Because there are no path from s to d directly link cost of s-d

is infinity by Dijkstra's algorithm. So, the shortest link was between s-r3.

After we go to r3, there are no links between r1 and r3; therefore, the value from the upper row is written into this cell again. The link cost between r3-r2 was 0.080814, and the link cost between r3-d was 0.000536. Since r3 couldn't connect to r3 again we wrote the same value as the cell in the above row. So, the shortest path was r3-d.

We wanted to send our message from node s to node d; therefore, our shortest path according to Dijkstra's algorithm is **s-r3-d**. The Dijkstra's Shortest Path table and the iterations to send message from node s to node d as explained before can be found below, as the Table 2: Dijkstra's Shortest Path :

TABLE II
DIJKSTRA'S SHORTEST PATH

	r1	r2	r3	d
s	0.121705	0.161641	0.000917	∞
r3	0.121705	0.080814	0.000917	0.000536
d	0.060932	0.080814	0.000536	0

This shortest path result can be confirmed by looking in the "configureR1.sh" and "configreR2.sh" files. These files add delay to the links of r1 and r2; therefore, these links have larger delay than the node r3's links as we obtained the results with the Dijkstra's table above.

D. Experiment

For experiment part, we first created new scripts for the nodes in the shortest path, s-r3-d path.

This time we declare s node as a server for r3 node, r3 node is client for s node and server for d node, and d node is the client of the r3 node. s node sends messages to r3 node and wait for feedback to send another message. r3 node takes messages from s node, give feedback to s node, send the message it gets from s node to d node than wait for the feedback of d node to repeat the actions. d node take the messages from r3 node than send a feedback to r3 node to show it is ready for another message.

To be able to calculate the end-to-end delay of the s-r3-d path, first thing we done is to take the time, *start=time.time()*, in s_exp.py and send it as a first message to the r3. after that we wait for feedback, then we sent 999 dummy message to the r3 to reach the desired message number. In r3_ex.py we take message, send feedback between r3 and s nodes and send message and take feedback between r3 and d nodes for 1000 times. In d_exp.py we take 1000 messages from r3 node and send feedback, also we which message that we received is the start time. We did it in a way that we sent "a" for every dummy message, if the message d node received was not "a" than it is the start time. After all 1000 messages reach to d node, we took time to know what is the ending time for the experiment. We calculated end-to-end delay by take the difference between ending and starting time.

Before starting to run our codes, we first synchronize our nodes using NTP. We did not need to synchronize our nodes in the discovery part because we take the difference of the times we take in the same node, but in the experiment part we take the difference of times we took from s node and d node.

To add the delays, we use tc command as
"**sudo tc qdisc add dev eth$x$ root netem delay _ms _ms distribution normal**".
"eth$x$" is depending on the paths and "ms" change for every experiment. To find which one we should use between eth0, eth1, eth2 and eth3, we used "ifconfig" command with the ip of paths. We found that s-r3 path uses eth2 and r3-d path uses eth3. The eth does not differ between s to r3 and r3 to s, or r3 to d and d to r3 even though ips are different. Because there are two paths that node r3 is connected, we used both
"**sudo tc qdisc add dev eth2 root netem delay _ms _ms distribution normal**".
and
"**sudo tc qdisc add dev eth3 root netem delay _ms _ms distribution normal**"
in every experiment for r3.

At the end of every experiment, we needed to clear the configurations to be able to add new ones. For that we used
"**sudo tc qdisc del dev eth$x$ root netem**"
command, again depending on the eth that paths have.

After we finished our implementations, we run our codes for each experiment and get the end-to-end delay. We used the following delays:

- For experiment 1: 20ms+-5ms
- For experiment 2: 40ms+-5ms
- For experiment 3: 50ms+-5ms

We get listed end-to-end delays for experiments:

- For experiment 1: 45.193471
- For experiment 2: 81.522314
- For experiment 3: 100.81955

For %95 confidence interval, the Z value is 1.960, so we calculated standard error with;
error = 1960*(deviation/math.sqrt(line_count)).
where line_count represents number of lines in the file. To calculate deviation we used built in function **statistics.stdev(end-to-end-delay)** .Our emulated delay versus end-to-end delay graph, which can be seen as Figure 2, is below:

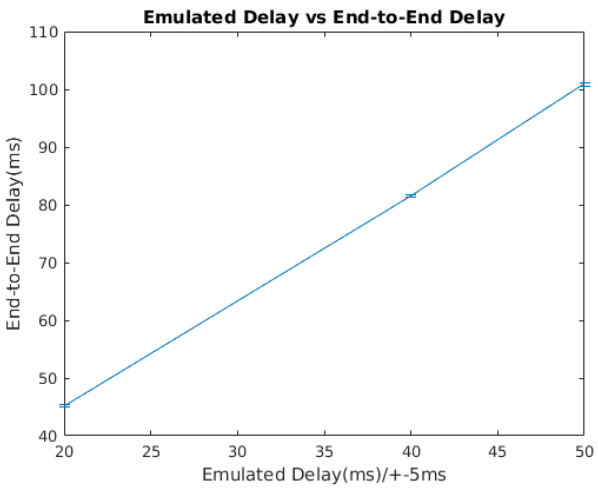

Fig. 2. Emulated Delay versus End-To-End Delay

We thought that the reason we get end-to-end delay nearly 2 times more than emulated delay is that, we used feedback to make sure every message we sent from s node reaches to r3 and d nodes. Even though we tried to not send feedback and get the same end-to-end delay with emulated delay, we could not even be sure that the starting time we send from s node is reaching to the d node. Hence we decided to use feedback messages.

As stated in the homework pdf, we created a folder named *experimentScripts* and put our experiment script o this directory. This script is used to send messages from s to d by following (s-r3-d) route.

E. Workshare

We implemented and designed all the parts together. Also, we wrote this report as a team.

III. Conclusion

For this project, first we created a slice by using given topology xml file. The topology has five nodes as s, r1, r2, r3 and d. The nodes on this topology employ the User Datagram Protocol, UDP,-based socket application. In this project we calculated the link cost between these nodes. We configured nodes r1 and r2 with given *configureR1.sh* and *configureR2.sh* files. When we look into these files we found that these files add extra delays to the links of r1 and r2 nodes. Therefore, we expected to obtain s-r3-d as our shortest path. When we created scripts for all of the nodes, as we expected we found out that s-r3-d route is the shortest path according to Dijkstra's shortest path algorithm.

For the experiment part of our project, we created new scripts for s, r3, and d nodes to find the end-to-end delay. After the experiment we saw that feedback messages also affects the end-to-end delay, and because we used feedback in our codes, end-to-end delay we found for every experiment is almost twice as much the emulated delay. We created a graph that shows emulated delay versus end-to-end delay to plot our solutions.

We submitted a folder called "discoveryScripts" which contains scripts to find link costs, and other folder "experimentScripts" which is used for experiment part and it sends a message to node d from node s. We also add our README file which explains the step to execute our codes.

References

[1] Popović, O. (2019, September 27). Dijkstra's Algorithm. Retrieved November 22, 2019, from https://stackabuse.com/graphs-in-java-dijkstras-algorithm/.