CENG435 Group 21 TP Part-1 Report

1st Zeki Ören
2264612

2nd Hasan Hüseyin Duykop
2035889

***Abstract*—This document includes the design and implementation of a UDP communication protocol over a network topology.**

***Index Terms*—node, RTT, end-to-end, emulation, link, script, socket, thread, Netem, NTP**

I. Introduction

This document covers the design and the implementation of a network communication protocol over a network topology that includes virtual machines and the links between them that are constructed via the GENI platform.

II. Design

A. Constructing Topology via the GENI Platform

GENI is a new, nationwide suite of infrastructure supporting ”at scale” research in networking, distributed systems, security, and novel applications. We will use the GENI platform to build our network topology.

After creating the slice in our project on GENI, we are going to build a network topology with the following .xml file. (Topogoly.xml). While reserving the resources we are going to set the site ”GENIC InstaGENI” option and load our Topology.xml file, and then reserve the resources.

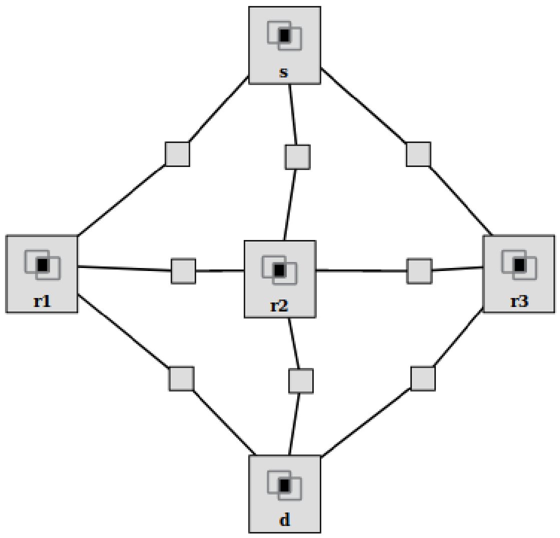

Fig. 1. Network topology.

Figure 1 shows the our network topology.

To connect to the nodes in our network topology, we will need to get an ssh key on the GENI platform. After getting our ssh key on GENI and making its configurations, we are going to connect to our nodes with the following command:

```
$ ssh −i <private key location>
<username>@hostname −p <port>
```

After connecting our nodes via the ssh, we are ready to configure our nodes.

B. Node Configurations

We are going to connect r_1 node and run the configuration file (ConfigureR1.sh). We are also going to connect r_2 node and run the configuration file (ConfigureR2.sh).

C. Approach to Calculate RTT Cost Between Nodes

RTT(Round Trip Time) is a time that is calculated between two nodes while they communicate. We are going to send a message from one node to another node, and we are going to get the same message from that node. We are going to get the time before sending the message, then we are going to get the time after we receive the same message from the node that we have sent the message. The difference between two times is going to be our RTT cost for that link.

We are going to have five scripts(one for each node), and we are going to have client/server architecture for each script.

Firstly, we will generate the messages on the r_2 node. The client, on r_2 node, is going to send these messages, and it's copies to s, r_1, r_3, and d nodes, and the servers on these four nodes are going to listen corresponding to ports to receive messages, that are sent from r_2 node. After that, the clients on s, r_1, r_3, and d nodes are going to send the same messages to r_2. In this way, r_2 is going to be able to calculate the RTT cost values of the links between r_2, and the other nodes, and save these costs in a file on the r_2 node.

After calculating the RTT costs on links between r_2, and the other nodes, we are going to do the same thing with r_1 and r_3 nodes. The clients on r_1 and r_3 are going to generate the messages, and the servers on s and d is going to listen corresponding to ports, and get these messages. After that, the clients on s and d is going to send the same messages to the servers on r_1 and r_3. In this way, r_1 and r_3 are going to be able to calculate the RTT cost values on links between s and d nodes, and save the costs in a file on r_1 and r_3 nodes, accordingly.

Since we need to do this concurrently, we are going to have different threads, that are running for each server and clients on each node. Because we need our servers and clients not to wait for others and affect the costs of the links.

On node s, we are going to have three threads. Each is going to run a server. One for r_1 node, one for r_2 node, and another one for r_3 node. These servers are also going to be able to send the same messages that are coming from r_1, r_2, and r_3, to each corresponding node. Similarly, the d node is going to have the same design with the s node. For r_1 and r_3 nodes, they are also going to have three threads. Two of them are running clients to send messages to s and d nodes, and one of them is running a server to receive messages from the r_2 node. For the r_2 node, there are going to be four threads. All of them are going to run servers to send messages to s, r_1, r_3, and d nodes.

III. Implementation

In this part, we are going to implement the scripts that will run our servers and clients in each node based on our design. For implementation, we are going to use the Python3 because properties of the Python3 is going to reduce our workload by using Python3 modules.

A. Node S Script

Since we are going to use UDP sockets, we have imported the Python3 module ”socket”, and we have also imported the ”threading” module so we can run our servers concurrently.

We have a function called ”threaded_ack” takes a socket as a parameter, and it does not return anything. This function uses sockets that we bind with corresponding hosts(r_1, r_2, r_3) and receives messages until there are no messages left. It also sends back the messages via the same sockets to corresponding hosts. In main, we have bound all three sockets, and we have started three threads with ”threaded_ack” function with the socket arguments. Then we use ”join()” function to wait our main for all threads to finish.

B. Node R1 Script

Additionally, this time, we have imported the ”time” module so that we can calculate the costs based on times.

We have a function ”threaded_ack” again. It takes a socket parameter and does not return anything. This is our server for receiving messages that come from r_2 node until there are no messages left. It also sends back the messages to the r_2 node via the same socket.

We also have a function ”threaded_send” that takes parameter socket and address, and it does not return anything. This is our client that will send messages to s and d nodes. It takes time just before sending the messages and, it also takes time just after receiving the messages from node s or d. It calculates the difference between start and end times. We do this with 100 messages and then calculate the average of them to find the average cost. Then, it writes the cost in a file ”link_cost.txt”.

In the ”main” function we assigned the sockets to r_2, s, and d nodes. Then, we started three threads one with ”threaded_ack” function with socket parameter r_2-r_1, and one with ”threaded_send” function with socket and host parameter for r_1-s, and another one with ”threaded_send” function with socket and host parameter for r_1-d. Finally, we use ”join()” to make our main for all threads to finish their jobs.

C. Node R2 Script

This time we have imported the same modules with R1 Script.

This time we have only ”threaded_send” function to send messages to nodes s, r_1, r_3, and d. It takes a parameter socket and address and does not return anything. This function the same as the ”threaded_send” function in the R2 script. It send 100 messages to other nodes and receives the same messages from other nodes and calculates their average cost then writes the results in the file called ”link_cost.txt”. This function is our client that are sending messages to other nodes.

In ”main” we have assigned the sockets and the host addresses to neighbor nodes(s, r_1, r_3, d). Then we have started four threads, one for each node with threaded_send function and corresponding sockets and addresses to other nodes. Then we used the ”join()” method to make the main wait for all threads to finish their jobs again.

D. Node R3 Script

We have imported the same modules with R1 and R2 Scripts.

This script similar to the R1 script. It has a ”threaded_ack” function that takes the socket parameter. It receives messages from the r_2 node and sends backs them to the r_2 node until there are no messages left.

It also has the ”threaded_send” function that takes the socket and address parameter and does not return anything as well as other scripts. Similar to others It sends 100 messages to neighbor nodes(s, d) and receive the same messages until there is no message left, and calculates the average cost then it writes these cost in a file called ”link_costs.txt”.

In ”main” we have assigned the sockets and the host addresses to neighbor nodes(s, d) and we bound to r_2 socket. Then we start three threads, one with ”threaded_ack” function with socket parameter r_2-r_3, one with ”threaded_send” function with socket and host parameter for r_3-s, and another one with ”threaded_send” function with socket and host parameter for r_3-d. Lastly, we use ”join()” to make our main for all threads to finish their jobs.

E. Node D Script

This script is similar to node S script. We have imported only ”socket” and ”threading” modules to use sockets and for communication and threads for concurrency.

We have a function called ”threaded_ack” that takes a socket as a parameter, and it does not return anything. This function uses sockets that we bind with corresponding hosts(r_1, r_2, r_3) and receives messages until there are no messages left. It also sends back the messages via the same sockets to corresponding hosts.

In main, we have bound all three sockets, and we have started three threads with ”threaded_ack” function with the socket arguments. Then, we use ”join()” function to wait our main for all threads to finish.

After finding the costs, we are going to need to find the shortest path.

F. Finding Shortest Path with Dijkstra Algorithm

Since we are going to have the costs after implementing our scripts that are going to calculate RTT costs between all the links in our network topology, we are going to use the Dijkstra Shortest Path Algorithm to make the shortest path table. The following figure is our network graph, and the costs are in milliseconds(ms).

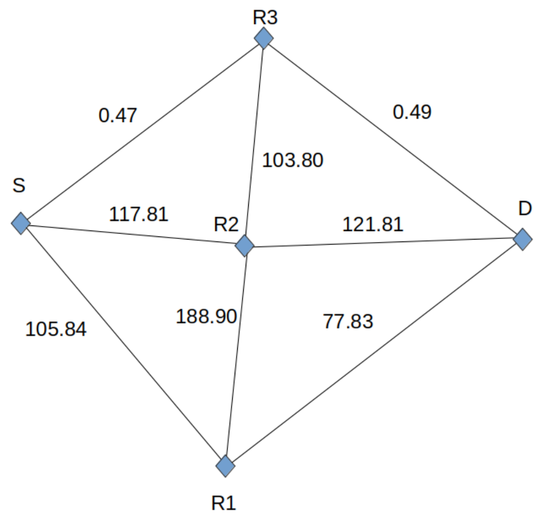

Fig. 2. Network graph

We have used the Dijkstra algorithm to find the shortest path. The shortest path we want to find is from source(s) to destination(d). By using this algorithm, we find the shortest-path as s-r_3-d path. Based on the above graph, we are going to make our Dijkstra table. As shown on the table, the cost of the shortest path from s to every node on the graph. For our path, we can easily say that the shortest path from source to destination costs 0.96 ms and the path is s-r_3-d.

The Dijkstra table is shown below:

	Unvisited Nodes	Visited Nodes	Current Node	s	r1	r2	r3	d
0	$\{s,r_1,r_2,r_3,d\}$	$\{-\}$	-	$(0,-)_0$	$(\infty,-)_0$	$(\infty,-)_0$	$(\infty,-)_0$	$(\infty,-)_0$
1	$\{r_1, r_2, r_3, d\}$	$\{s\}$	s		$(105.84,s)_1$	$(117.81,s)_1$	$(0.47,s)_1$	$(\infty,s)_1$
2	$\{r_1, r_2, d\}$	$\{s,r_3\}$	r_3		$(105.84,s)_1$	$(104.28,r_3)_2$		$(0.96,r_3)_2$
3	$\{r_1, r_2\}$	$\{s,r_3, d\}$	d		$(78.79,d)_3$	$(104.28,r_3)_2$		
4	$\{r_2\}$	$\{s,r_1, r_3, d\}$	r_1			$(104.28,r_3)_2$		
5	$\{-\}$	$\{s,r_1, r_2, r_3, d\}$	r_2					

Fig. 3. Shortest Path with Dijkstra Algorithm

IV. Experiments

In this part, we know that the shortest path from source(s) to destination(d) is the path s - r_3 - d. We have found this case by using the Dijkstra Algorithm.

After this part, we tested our link to see that the source can send data completely to do destination. Then, the next step was to create the network emulation delay and the end-to-end delay with a 95 percent confidence interval.

A. Node S Script

We have imported the Python3 module ”socket” and we have also imported the NTP library(ntplib) to establish the synchronization of the nodes.

In this script, we sent 100 messages to node r_3. This message is the time of the r_3 node as the starting time of the end-to-end delay. This script is our client that are sending messages to the rooter.

B. Node R3 Script

We have imported two Python3 module ”socket”s. We bound one of the sockets to listen to s script, and the other one is to send the same message, which has received from s to the d.

In this script, we sent the same 100 messages, which have received from the node s to node d. This script is our rooter that is directing messages to the destination.

C. Node D Script

We have imported the Python3 module ”socket”. We bound the socket to listen R3 script.

In this script, we receive 100 messages, which have sent from the node s to node d. As we know that, the messages, which came from the source, consist of the time of the r_3 node when the message sent from the s node. Then, in this script, the time of received from the s and the time of r_3 when the data arrive at the d node are used to calculate the end-to-end delay of the link. We use the time of the node r_3 to synchronize the time. To do this, we also imported the NTP library(ntplib) in this node.

After the calculation of the costs finished, the script creates a file to save the cost simultaneously. The name of this file is ”delay_exp#.txt”. The # is changing for all experiments. This script is our server that is receiving messages from the source.

D. Netem Commands

After selecting the shortest path, we need to find the end-to-end delay of it. As we know, we have previously configured r_1 and r_2 nodes. We had to configure the r_3 node in the test phase because there were some experiments we needed to do for the end-to-end delay calculations.

The first experiment is 20 ± 5 ms, second is 40 ± 5 ms, and the last one is 50 ± 5 ms delay. To establish these delays, we used the Netem commands.

These commands establish some delays in packet delivery. Also, we aim to find a relation between end to end delay and network emulation delay by using this information.

Firstly, we run the scripts on the s, r_3, and d to send some data from source to destination without using Netem commands. By the way, we got an end-to-end delay of this path without using any network emulation delay. We applied the same steps 25 times, repetitive. Then, we took the mean of these 25 values and obtained a single value, which gives average end-to-end delay s to destination in this path.

Then, configured r_3 with the Netem commands to establish network emulation delay in the first experiment. We have repeated this process 25 times too and we have got the mean of the results and obtained a single value which gives average end-to-end delay with 20 ± 5 ms network emulation delay.

We have repeated the process, which I explained in the paragraph above with using experiments 2 and 3 (40 ± 5 ms and 50 ± 5 ms delays).

E. Average Script in Node D

This script calculates the mean of the data in a file. As we explained before, the d script creates a file, whose name is like ”delay_exp#.txt”. Because, we have 4 experiments, we created four .txt files, which are ”delay_exp0.txt”, ”delay_exp1.txt”, ”delay_exp2.txt”, ”delay_exp3.txt”.

In this script, we calculate the mean of these experiments. This script is a helper for our calculations.

As a result, we obtained 4 different end-to-end delays to use in our chart. Here is the chart which shows the relation between end to end delay and network emulation delay:

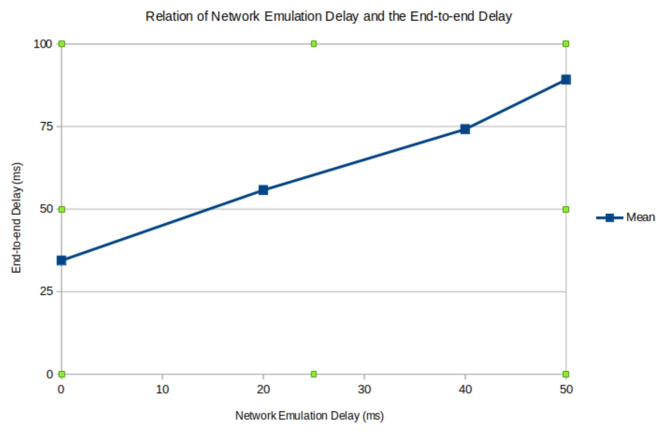

Fig. 4. Relation Between End-to-end Delay and Network Emulation Delay

In this graph, we calculated the end-to-end delay with network emulation delay in the links between source(s) and destination(d). We implemented 20 ± 5 ms, (40 ± 5 ms and 50 ± 5 respectively. As shown in the table above, we can easily say that some difference delays occur with different emulations. This difference affects the end-to-end delay of the link slightly because the propagation delay was not changed while processing delay is increased.