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.

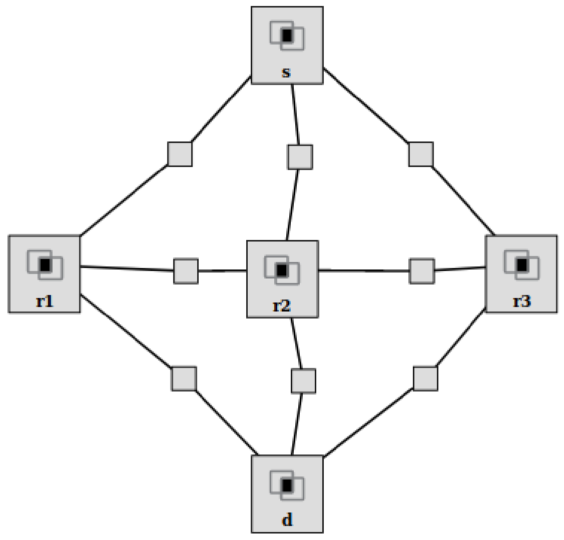


Fig. 1. Network topology.

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>
```

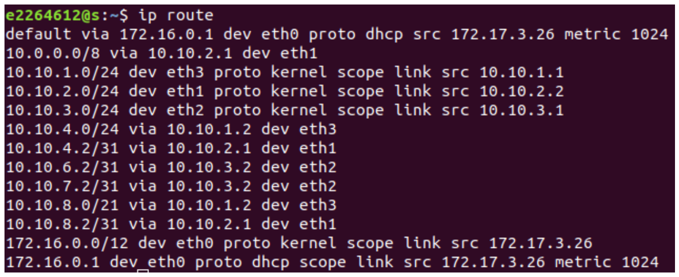
After connecting our nodes via the ssh, we are ready to configure our nodes.

B. Node Link Configurations

As you can see in the design each node has multiple links to the other nodes. When we decide to send a packet through nodes, the packet can follow multiple ways. But in our design we want to send packets by using specific links. To establish this, we need to close some links so the packet can follow the links which we desire. Firstly, we controlled every nodes to see which links they have. The see the links from each node we used the following command in the nodes:

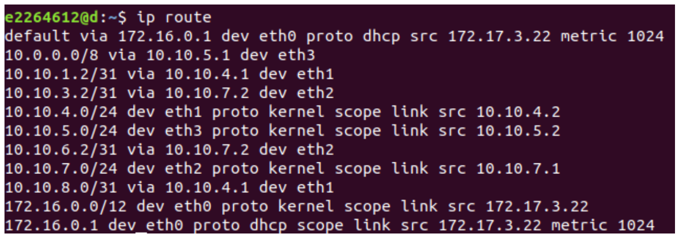
```
ip route
```

When we use this command we saw that in the nodes:



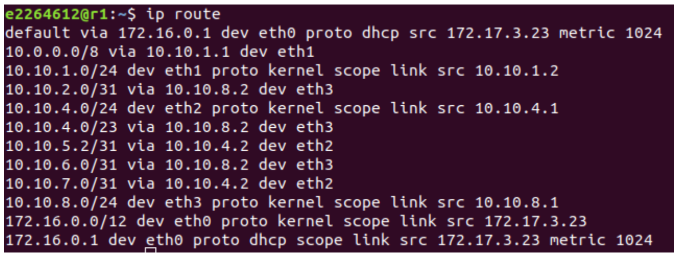
```
e2264612@s:~$ ip route
default via 172.16.0.1 dev eth0 proto dhcp src 172.17.3.26 metric 1024
10.0.0.0/8 via 10.10.2.1 dev eth1
10.10.1.0/24 dev eth3 proto kernel scope link src 10.10.1.1
10.10.2.0/24 dev eth1 proto kernel scope link src 10.10.2.2
10.10.3.0/24 dev eth2 proto kernel scope link src 10.10.3.1
10.10.4.0/24 via 10.10.1.2 dev eth3
10.10.4.2/31 via 10.10.2.1 dev eth1
10.10.6.2/31 via 10.10.3.2 dev eth2
10.10.7.2/31 via 10.10.3.2 dev eth2
10.10.8.0/21 via 10.10.1.2 dev eth3
10.10.8.2/31 via 10.10.2.1 dev eth1
172.16.0.0/12 dev eth0 proto kernel scope link src 172.17.3.26
172.16.0.1 dev eth0 proto dhcp scope link src 172.17.3.26 metric 1024
```

Fig. 2. S Links.



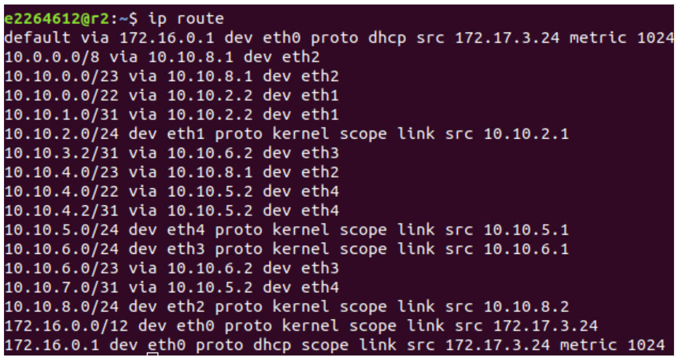
```
e2264612@d:~$ ip route
default via 172.16.0.1 dev eth0 proto dhcp src 172.17.3.22 metric 1024
10.0.0.0/8 via 10.10.5.1 dev eth3
10.10.1.2/31 via 10.10.4.1 dev eth1
10.10.3.2/31 via 10.10.7.2 dev eth2
10.10.4.0/24 dev eth1 proto kernel scope link src 10.10.4.2
10.10.5.0/24 dev eth3 proto kernel scope link src 10.10.5.2
10.10.6.2/31 via 10.10.7.2 dev eth2
10.10.7.0/24 dev eth2 proto kernel scope link src 10.10.7.1
10.10.8.0/31 via 10.10.4.1 dev eth1
172.16.0.0/12 dev eth0 proto kernel scope link src 172.17.3.22
172.16.0.1 dev eth0 proto dhcp scope link src 172.17.3.22 metric 1024
```

Fig. 3. D Links.



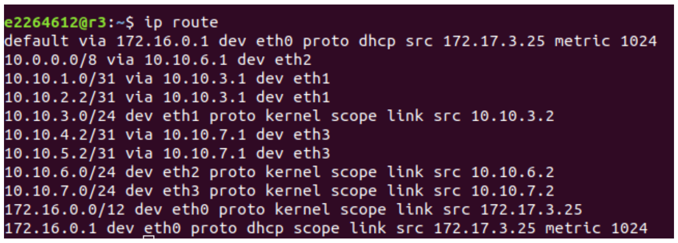
```
e2264612@r1:~$ ip route
default via 172.16.0.1 dev eth0 proto dhcp src 172.17.3.23 metric 1024
10.0.0.0/8 via 10.10.1.1 dev eth1
10.10.1.0/24 dev eth1 proto kernel scope link src 10.10.1.2
10.10.2.0/31 via 10.10.8.2 dev eth3
10.10.4.0/24 dev eth2 proto kernel scope link src 10.10.4.1
10.10.4.0/23 via 10.10.8.2 dev eth3
10.10.5.2/31 via 10.10.4.2 dev eth2
10.10.6.0/31 via 10.10.8.2 dev eth3
10.10.7.0/31 via 10.10.4.2 dev eth2
10.10.8.0/24 dev eth3 proto kernel scope link src 10.10.8.1
172.16.0.0/12 dev eth0 proto kernel scope link src 172.17.3.23
172.16.0.1 dev eth0 proto dhcp scope link src 172.17.3.23 metric 1024
```

Fig. 4. R1 Links.



```
e2264612@r2:~$ ip route
default via 172.16.0.1 dev eth0 proto dhcp src 172.17.3.24 metric 1024
10.0.0.0/8 via 10.10.8.1 dev eth2
10.10.0.0/23 via 10.10.8.1 dev eth2
10.10.0.0/22 via 10.10.2.2 dev eth1
10.10.1.0/31 via 10.10.2.2 dev eth1
10.10.2.0/24 dev eth1 proto kernel scope link src 10.10.2.1
10.10.3.2/31 via 10.10.6.2 dev eth3
10.10.4.0/23 via 10.10.8.1 dev eth2
10.10.4.0/22 via 10.10.5.2 dev eth4
10.10.4.2/31 via 10.10.5.2 dev eth4
10.10.5.0/24 dev eth4 proto kernel scope link src 10.10.5.1
10.10.6.0/24 dev eth3 proto kernel scope link src 10.10.6.1
10.10.6.0/23 via 10.10.6.2 dev eth3
10.10.7.0/31 via 10.10.5.2 dev eth4
10.10.8.0/24 dev eth2 proto kernel scope link src 10.10.8.2
172.16.0.0/12 dev eth0 proto kernel scope link src 172.17.3.24
172.16.0.1 dev eth0 proto dhcp scope link src 172.17.3.24 metric 1024
```

Fig. 5. R2 Links.



```
e2264612@r3:~$ ip route
default via 172.16.0.1 dev eth0 proto dhcp src 172.17.3.25 metric 1024
10.0.0.0/8 via 10.10.6.1 dev eth2
10.10.1.0/31 via 10.10.3.1 dev eth1
10.10.2.2/31 via 10.10.3.1 dev eth1
10.10.3.0/24 dev eth1 proto kernel scope link src 10.10.3.2
10.10.4.2/31 via 10.10.7.1 dev eth3
10.10.5.2/31 via 10.10.7.1 dev eth3
10.10.6.0/24 dev eth2 proto kernel scope link src 10.10.6.2
10.10.7.0/24 dev eth3 proto kernel scope link src 10.10.7.2
172.16.0.0/12 dev eth0 proto kernel scope link src 172.17.3.25
172.16.0.1 dev eth0 proto dhcp scope link src 172.17.3.25 metric 1024
```

Fig. 6. R3 Links.

So, we know that in experiment 1 the packets will follow s -¿ r3 -¿ d because we calculated the shortest path by using the Dijkstra Algorithm. So when we decide to send a packet in experiment 1, the path must be this one only.

In the experiment 2 the packets will follow s -¿ r1 -¿ d or s -¿ r2 -¿ d. So when we decide to send a packet in experiment 2, the paths must be these only.

As a result, we closed redundant links in the s and d nodes by using these commands:

```
sudo ip link set [INTERFACE] down
sudo ip link set [INTERFACE] up
```

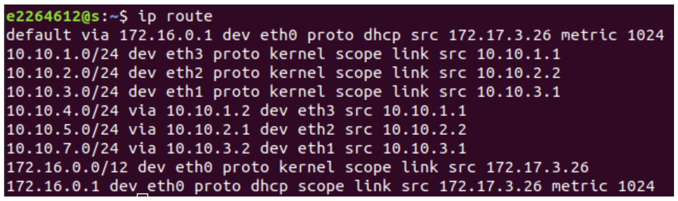
In s node we closed all interfaces then added needed links. In the node s, we use these commands:

```
sudo ip link set eth1 down
sudo ip link set eth1 up
sudo ip link set eth2 down
sudo ip link set eth2 up
sudo ip link set eth3 down
sudo ip link set eth3 up
sudo ip route add 10.10.4.0/24 via 10.10.1.2 src
sudo ip route add 10.10.5.0/24 via 10.10.2.1 src
sudo ip route add 10.10.7.0/24 via 10.10.3.2 src
```

In the node d, we use these commands:

```
sudo ip link set eth1 down
sudo ip link set eth1 up
sudo ip link set eth2 down
sudo ip link set eth2 up
sudo ip link set eth3 down
sudo ip link set eth3 up
sudo ip route add 10.10.1.0/24 via 10.10.4.1 src
sudo ip route add 10.10.2.0/24 via 10.10.5.1 src
sudo ip route add 10.10.3.0/24 via 10.10.7.2 src
```

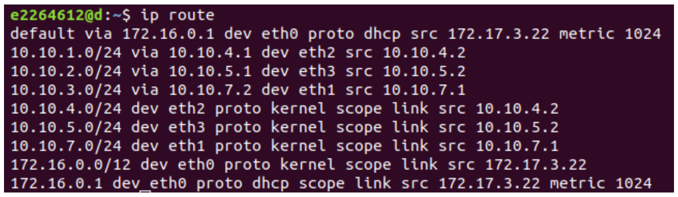
So the final version of the s node is:



```
e2264612@s:~$ ip route
default via 172.16.0.1 dev eth0 proto dhcp src 172.17.3.26 metric 1024
10.10.1.0/24 dev eth3 proto kernel scope link src 10.10.1.1
10.10.2.0/24 dev eth2 proto kernel scope link src 10.10.2.2
10.10.3.0/24 dev eth1 proto kernel scope link src 10.10.3.1
10.10.4.0/24 via 10.10.1.2 dev eth3 src 10.10.1.1
10.10.5.0/24 via 10.10.2.1 dev eth2 src 10.10.2.2
10.10.7.0/24 via 10.10.3.2 dev eth1 src 10.10.3.1
172.16.0.0/12 dev eth0 proto kernel scope link src 172.17.3.26
172.16.0.1 dev eth0 proto dhcp scope link src 172.17.3.26 metric 1024
```

Fig. 7. Essential Links on S Node.

So the final version of the d node is:



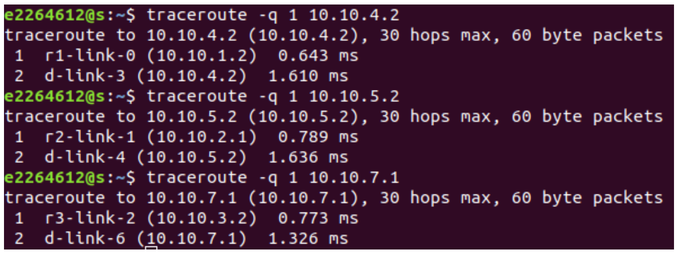
```
e2264612@d:~$ ip route
default via 172.16.0.1 dev eth0 proto dhcp src 172.17.3.22 metric 1024
10.10.1.0/24 via 10.10.4.1 dev eth2 src 10.10.4.2
10.10.2.0/24 via 10.10.5.1 dev eth3 src 10.10.5.2
10.10.3.0/24 via 10.10.7.2 dev eth1 src 10.10.7.1
10.10.4.0/24 dev eth2 proto kernel scope link src 10.10.4.2
10.10.5.0/24 dev eth3 proto kernel scope link src 10.10.5.2
10.10.7.0/24 dev eth1 proto kernel scope link src 10.10.7.1
172.16.0.0/12 dev eth0 proto kernel scope link src 172.17.3.22
172.16.0.1 dev eth0 proto dhcp scope link src 172.17.3.22 metric 1024
```

Fig. 8. Essential Links on D Node.

I used the following command to see the packet which I sent to reach the expected node with the expected path:

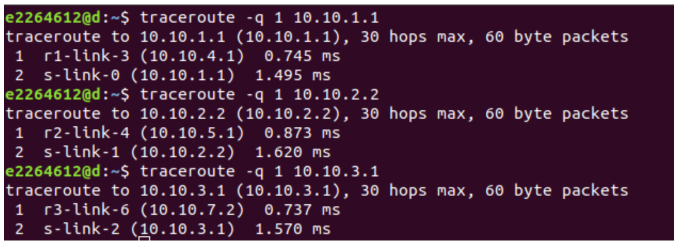
```
traceroute −q [Number of Packets] [IP adress]
```

The outputs are as followings:



```
e2264612@s:~$ traceroute -q 1 10.10.4.2
traceroute to 10.10.4.2 (10.10.4.2), 30 hops max, 60 byte packets
 1  r1-link-0 (10.10.1.2)  0.643 ms
 2  d-link-3 (10.10.4.2)  1.610 ms
e2264612@s:~$ traceroute -q 1 10.10.5.2
traceroute to 10.10.5.2 (10.10.5.2), 30 hops max, 60 byte packets
 1  r2-link-1 (10.10.2.1)  0.789 ms
 2  d-link-4 (10.10.5.2)  1.636 ms
e2264612@s:~$ traceroute -q 1 10.10.7.1
traceroute to 10.10.7.1 (10.10.7.1), 30 hops max, 60 byte packets
 1  r3-link-2 (10.10.3.2)  0.773 ms
 2  d-link-6 (10.10.7.1)  1.326 ms
```

Fig. 9. Traceroute S to D.



```
e2264612@d:~$ traceroute -q 1 10.10.1.1
traceroute to 10.10.1.1 (10.10.1.1), 30 hops max, 60 byte packets
 1  r1-link-3 (10.10.4.1)  0.745 ms
 2  s-link-0 (10.10.1.1)  1.495 ms
e2264612@d:~$ traceroute -q 1 10.10.2.2
traceroute to 10.10.2.2 (10.10.2.2), 30 hops max, 60 byte packets
 1  r2-link-4 (10.10.5.1)  0.873 ms
 2  s-link-1 (10.10.2.2)  1.620 ms
e2264612@d:~$ traceroute -q 1 10.10.3.1
traceroute to 10.10.3.1 (10.10.3.1), 30 hops max, 60 byte packets
 1  r3-link-6 (10.10.7.2)  0.737 ms
 2  s-link-2 (10.10.3.1)  1.570 ms
```

Fig. 10. Traceroute D to S.

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. We created scripts for s and d nodes only, because we can send packets through r_1, r_2 and r_3 routers by using sockets in the s and d nodes.

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. Moreover, we imported ”hashlib”. The ”hashlib” library is a library that allows us to encrypt our data in different algorithms. We use it to calculate checksum. Then, we imported ”pickle” to convert data to the byte string.

In the main function we opened the ”input.txt” file to read and save it onto a variable ”file_data”, then closed it. For experiment 1 we printed the ”Starting experiment 1...” to see is it working. Then, we created a socket that represents r_3 to d socket.

There are 10000 messages on our design, and the size of each message is 500 bytes. we created a message which is added from ”file_data”. The message is 500 bytes long. Then calculate the checksum using ”hashlib”, and create a packet with this checksum and message in the makePacket function which creates a header with sequence number and checksum then, the packet with this header and message.

In the design, we are sending one packet at a time, and waiting for an acknowledge. If the acknowledge is not equal to the sequence number, we are sending the same packet again, because we want to avoid packet loss. If they are equal we are increasing the sequence number, and sending the next packet.

For experiment 2, we use 2 threads to send 2 data at a time by using different sockets which are r_1 and r_2. We created a send function that gets the data, socket name and address of socket parameters. This function is using threads simultaneously. In the function, we use the same method to send the packet as experiment 1, except lock the thread to wait until it sends the packet.

B. Node D Script

In this script, we generally get the data, send an acknowledge message to the s node. Also, we have imported the same ones in the s.py script.

For experiment 1, we are creating an ”output1.txt” file to save the data which are coming from the s node. Then, we create a socket and bind it to be able to listen the socket. In a forever while loop we listen the socket, get the data which the s.py sent, and save it to the message parameter. If the message is not null, we get the packet from message, sequence number from packet, checksum from packet, and data from packet. If the checksum is equal to the calculated checksum, we send as acknowledge number by adding one to the sequence number. Then, we write the data to the ”output1.txt” file. If the checksum is not equal to the calculated checksum, we send the same acknowledge number to avoid packet loss. When all the data came in, we stopped the script.

For experiment 2, we are creating an ”output2.txt” file to save the data which are coming from the s node. Then, we use 2 threads to get 2 data at a time by using different sockets which are r_1 and r_2. We created a receive function that gets the socket name and the ”output2.txt” file. This function is using threads simultaneously. In the function, we use the same method to get the packet as experiment 1, except lock the thread to wait until it sends the packet.