CENG 435 - Data Communications and Networking Term Project - Part II

Sina Sehlaver
Department of Computer Engineering
Middle East Technical University
2099729

Beyazit Yalcinkaya
Department of Computer Engineering
Middle East Technical University
2172138

***Abstract*—In this project, we present our own reliable data transfer protocol design that is built on top of UDP. A modified version of selective repeat is implemented, pipelining method is used, multi-homing is practiced, single-ACKs are used, and an auto-link-down detection is developed. The performance of this reliable data transfer protocol of our own is measured with two experiments. The first experiment examined the performance through one link, whereas the second experiment showed the effects of using multi-homing with two links. As a result, we concluded that in both experiments the increase in packet loss increases the file transfer time considerably and creates an overhead for the performance. Furthermore, the comparison between the two experiments point out the benefits of using multi-homing in order to increase the utilisation of links and shorten file transfer time.**

I. Introduction

This paper explains the details of the second part of the term project. In this part of the term project, we are expected to send a file with exactly five Mb file size from source to destination with a self-designed reliable data transfer protocol on top of UDP, in a given network topology. Our motivation was to get familiar with the reliable data transfer process and its challenges. In accordance with our motivation, we have designed our own reliable data transfer protocol that includes our combination of modified methods such as selective repeat, pipelining and multi-homing. On top of these, we have created our own methods such as auto-link-down detection. Since the experiment is extremely case-specific as in we will always send exactly five Mbs (5000000 bytes), we have implemented our reliable data protocol in accordance with these specifications. Our protocol, transfers the file in a reliable fashion, as promised.

Multi-homing is one of the important aspects of our reliable data protocol. Multi-homing is implemented in a fashion that it splits the file into two parts and sends them separately. We have implemented the protocol in a way that in the case of one of the routers being down (at most one can be down), it will redirect the file transmission traffic onto the other "alive" router so that the reliable data transfer is not interrupted. In order to examine the performance of our protocol from multiple view-points, we have conducted two experiments. The first one is conducted in a way that it transfers the file through one router, that is $r3$. The latter one, on the other hand, transfers the file through two distinct routers, namely $r1, r2$. Both of them measure the performance of our reliable transfer protocol, however the second one also measures the multi-homing method's performance.

For both of the experiments, we have used the "tc/netem" setup configuration in order to emulate packet loss with delays on the links. We have specified the packet loss percentages and delay durations when calling "tc/netem". With the help of the emulation, we were able to observe the relationship between packet loss percentage and file transfer time. Since our packet based protocol is practicing reliable data transfer, each packet must arrive to the destination and thus, packet loss directly affects the file transfer time.

When we compare the two experiments, we see that multi-homing has a positive impact on the performance of our protocol since it reduces the file transfer time, as expected. We have repeated our experiments such that the error margin of our results did not exceed 0.1 with a confidence interval of 95% and gathered the data with this statistical assurance.

The outline of our paper is as follows; in the second section we will explain the methodologies we have used in these experiments, in the third section we give specific implementation details and justify them, in the fourth section we report and discuss our experimental results and section 5 concludes the paper.

II. Methodology

This section describes the design and logic of the methods used in the experiments. We have reserved resources through the GENI platform with the topology described inside the XML format topology file. We have used the specific IP addresses configured by this file to connect each node in our topology to each other.

In accordance with the specifications given to us, that is sending a file from s node to d node with exactly five Mbs file size in the given network topology, we have designed a protocol which will ensure reliable data transfer. We have implemented pipelining, multi-homing and auto-link-down detection. We have setup two experiments to test our reliable data transfer protocol. The first one aimed to measure the performance of the protocol with respect to the packet loss through one path whereas the second one used two distinct paths while sending the data in order to measure the effects of multi-homing on the performance.

Code Segment for Multi-homing, Pipelining and Automatic Link Down Detection

```
# Function that is responsible from reliabley tranfering a given packet
def sender(file_sender_socket, i):
    global dst_ips, payload_size, N, base, window_size, is_acked, is_sent, byte_chunks, timeout_interval, estimated_rtt
    global dev_rtt, timeout_interval, estimated_rtt, timeout_interval_lock, starts, is_timedout, is_timedout_lock, timeouts
    send_count = 0# how many times packet is sent to indicate link's down/up status
    key = i % 2 if dst_ips[0] != dst_ips[1] else 0# calculate key, that is index
    f = False# send_count did not exceed 1000 yet
    while not is_acked[i]:# while packet with given index is not acked yet
        is_timedout_lock.acquire()# Acquire lock for timeout checks
        if send_count >= 1000:# if send_count exceeeds 1000
            is_timedout = True# set flag
            timeouts.append(i)# append index of packet to the failed packets list
            f = True# set flag
        is_timedout_lock.release()# Release lock
        if f:# if flag is set, return
            return
        timeout_interval_lock[key].acquire()# Acquire key for fetching timeout value
        timeout_value = timeout_interval[key]# store timeout value
        timeout_interval_lock[key].release()# Release lock
        starts[i] = time.time()# save start time of packet for rtt
        file_sender_socket.sendto(create_packet(i + 1, 0, byte_chunks[i]), (dst_ips[key], 8080))# send packet
        time.sleep(timeout_value)# sleep until timeout value, after wake up while loop checks if the ack is received
        send_count += 1# increment send_count
```

In these experiments, in order to configure specific packet loss percentages that are given to us, we have used "tc/netem" command. This command enabled us to emulate packet loss (with a specified rate), during the transmission of packets between the nodes in the network. Our reliable data transfer protocol repeatedly sent the lost packets in order to compensate the emulated loss. Through experimentation, we have observed that this repeated transmission, and therefore, the packet loss had a negative effect on the file transfer time.

For pipelining, with an inspiration from selective repeat, we have created our own method. Just as it is with selective repeat; our sender, sends the packets that are within a window with a pre-determined size in a pipelined fashion, that is, it sends two consecutive packets without waiting for the arrival of their corresponding ACK messages. Each packet inside this window has a timeout duration that is calculated dynamically in accordance with the network's state and during this timeout period, the sender waits for their ACK messages to arrive. In case of an ACK message not arriving during the timeout duration, it will resend the packet. For the packets that received their ACK messages, we have increased the "base" such that none of the packets that are not ACKed would be left outside of the window.

For the multi-homing method, we have split the file into two, in an abstract fashion. The packets that have odd indexes used one distinct path while the even indexed packets used the other path. This division of file to paths, actually enabled a faster transmission of the file to the destination without exhausting the paths. For the case of "at most one of the links being down" in this multi-homing part, we have come up with a method that prevents the interruption of the file transfer. Our method is as follows; if a packet does not receive its ACK message during its timeout period, a 1000 consecutive times, we deduce that that link is "down" and we redirect the file transmission to the other link that we believe is up.

We did not use any "NAK" messages in our implementation but only "ACK" messages. For the buffering, we used global arrays inside the python script of node d that holds the received byte chunks. The case of failure in receiving a packetm is handled with timeouts. The timeout value is calculated dynamically after each packet transmission by using following equations.

$$\begin{aligned} EstimatedRTT &= (1-\alpha)EstimatedRTT \\ &\quad + \alpha SampleRTT \\ DevRTT &= (1-\beta)DevRTT \\ &\quad + \beta|SampleRTT - EstimatedRTT| \\ TimeoutInterval &= EstimatedRTT + 4DevRTT \end{aligned} \tag{1}$$

where we used $\alpha = 0.125$ and $\beta = 0.25$.

For the calculation of round-trip time (RTT); for each packet we save the time just before we send it and when the corresponding ACK message of the packet is received, we calculate the time between these two events and that becomes our $SampleRTT$. After the calculation of $SampleRTT$ we apply the equations above to get the $TimeoutInterval$. For the distinct paths in our second experiments, we did the calculations for each of them separately.

The packets consist of a header and a payload.

Header consists of;

- Sequence number (2 bytes)
- ACK number (2 bytes)
- Length (2 bytes)
- Checksum (32 bytes)

Payload consists of 512 bytes of data chunk.

For the checksum, we have used the "md5" sum method. It checks whether the payload (and only the payload) is received without any corruption. It is a well-known and reliable method that is widely-used.

Margin of Error Tables

Experiment 1

Loss Percentage	Mean	Margin of Error
5 %	48.150447404	9.9696 %
15 %	81.197790027	1.5739 %
28 %	172.959200991	8.8248 %

Table 1

Experiment 2

Loss Percentage	Mean	Margin of Error
5 %	41.3456017956	9.9737 %
15 %	73.1446991897	9.9973 %
28 %	161.939675249	9.9912 %

Table 2

When all packets receive their corresponding ACK messages, sender sends a packet with sequence and ACK number both 0 in a reliable fashion (resends it until the arrival of a corresponding ACK message) in order to inform the receiver that the transmission is done. When receiver receives this message, as opposed to the sender, unreliably sends a packet with the same configuration (seq and ACK no 0) and terminates. If the sender receives this new packet sent from the receiver, it terminates safely. However, in the case of not receiving this new packet, the sender continues to try to send the same packet over and over until it sends it 1000 consecutive times that result in timeout.

After that the file transmission is done with the above mentioned methods and conventions, sender reports the file transfer time. This file transfer experiment is conducted repeatedly until the results ensure a margin of error less than 0.1 with a confidence interval of 95%. The number of repetition is also calculated dynamically, that is, we calculate the margin of error after each new experiment result and stop the execution when the conditions are satisfied. The reported numbers are discussed in the experiment results section.

III. Implementation

This section of the report, explains the implementation details of the methodology section. We used Python3 and Bash scripts, each Python3 script "main.py" is run with a Bash script "start.sh" for each corresponding node.

Several modules were used:

- socket module for the transmission
- threading module for concurrency
- struct module for packet header construction
- sys module for input arguments
- time module for round trip time, file transfer time and timeout
- hashlib module for md5 calculation
- statistics for statistical tools
- math for mathematical tools
- tc/netem command to emulate network packet loss and delay

We have created a "README.md" file, explaining the structure of the scripts and the instructions for the reproduction of the experiment environment. Each script also include comments that elaborate on the functionality.

$r1, r2, r3$ nodes run a simple python script that includes a routing table for the correct redirection of the packets and uses this routing table to send the arriving packets to their destinations. This python script is called from the bash script "start.sh". This bash script calls other bash scripts that configure the network settings of each connection of that node, respectively and in order: reset, initialisation and configuration. The script "start.sh" takes loss percentage as argument.
These setting scripts use "tc/netem" commands:

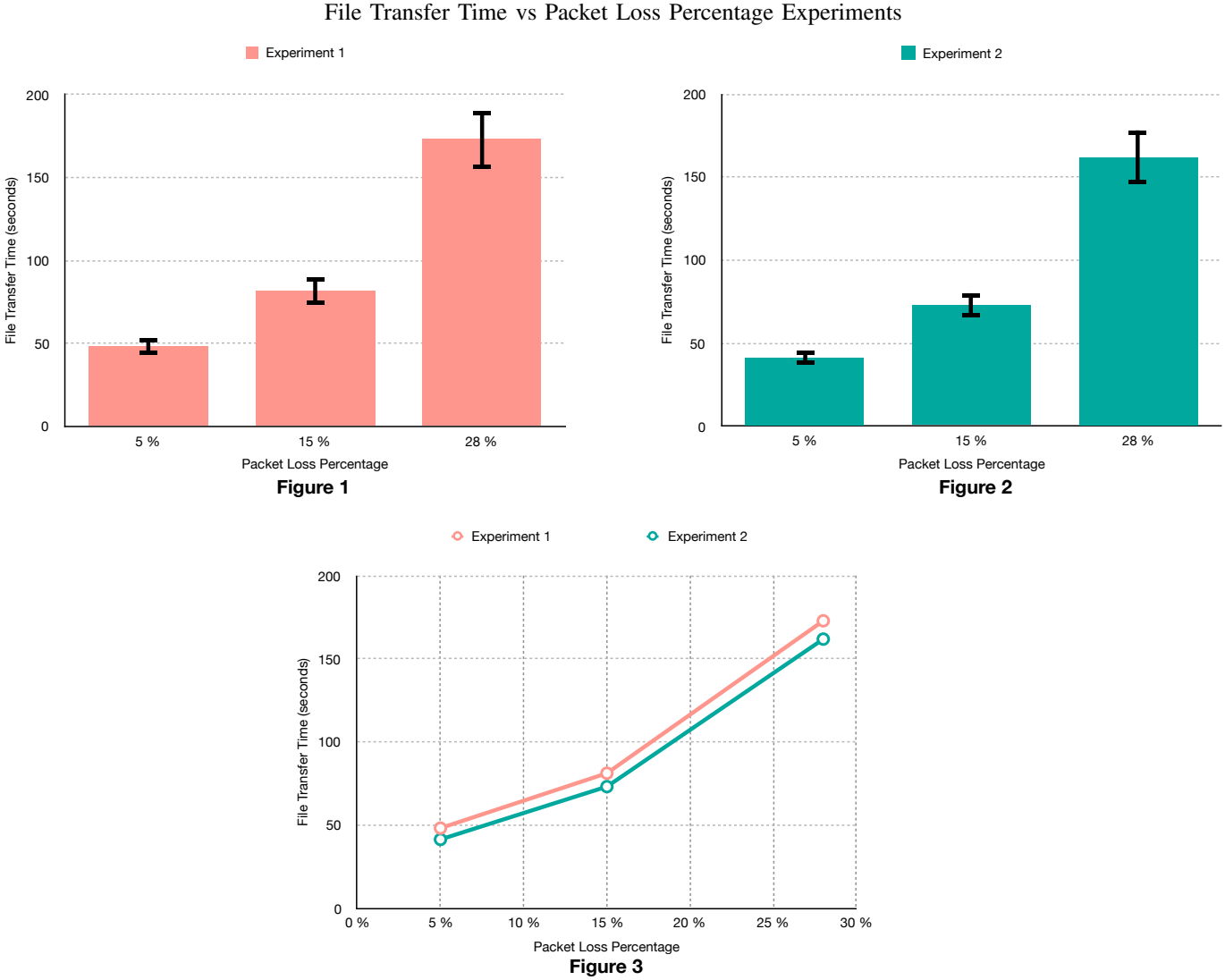
- first reset all the previous settings with "del" command
- second initialise with "add" command
- finally configure loss percentages and delays with "change" command

Node s also has the same reset, init and config scripts that $r1, r2, r3$ have. Alongside with those, it has a "main.py" that is different from all others and a "start.sh" which also has a different implementation from all others.
The script "start.sh" accepts the following arguments as its input arguments:

- Experiment type
- Loss percentage
- Input file name
- Margin of error threshold

After calling the setting bash scripts mentioned above, the "start.sh" script calls "main.py" with input arguments; experiment type, input file name and margin of error threshold. "main.py" checks whether margin of error threshold is -1, if so, it deduces that the experiment is supposed to run only once. On the other hand if it is a non-negative value, the experiment is repeated until the margin of error falls under that threshold.

The script "main.py" in s splits the file into byte chunks and then creates an "ACK receiver thread". When an ACK message is received, this thread marks the packet's ACK message as "received" inside a global array that holds the states of ACK messages of all packets. At the same time, when an ACK message is received, this receiver thread calculates the corresponding packet's round trip time and then does the calculations mentioned in the methodology section to fetch the timeout value. After these, "file sender" function is called. This function, creates a special "sender" thread in order to send each byte chunk with the modified selective repeat method that we have mentioned above. The sender thread takes a specific



byte chunk, creates a packet such that this byte chunk is its payload and then reliably sends it to the destination.
In order to reliably send the file, while the packet's ACK message is not received;

- It sends the packet
- sleeps for the $timeout$ duration

If the ACK message is not received by the end of the sleep, it re-sends the packet. This re-transmission is counted as $timeout$ and if a packet is re-sent 1000 times because of $timeout$, the file sender function is informed via a global variable (*Code segment figure*) so that the file sender function can detect it in its main loop and redirect its remaining file transfer to the link that is up. When all ACK messages arrive to s, s sends a packet with sequence and ACK number 0 to d so that d is informed that the file transmission is over.

Node d also has the same reset, init and config scripts that $r1, r2, r3$ have. Alongside with those, it has a "main.py" that is different from all others and a "start.sh" which also has a different implementation from all others.

The script "start.sh" accepts the following arguments as its input arguments:

- Loss percentage
- Output file name
- Repetition count

After calling the setting bash scripts mentioned above, the "start.sh" script calls "main.py" with input arguments; output file name and repetition count. "main.py" checks whether repetition count is -1, if so, it runs in an infinite loop. On the other hand if it is a non-negative value, the experiment is repeated as many times as the value. "main.py" calls the file receiver function. This function checks the checksums of received packets. If they are intact and the file receiver function did not receive another packet with the same sequence number before, it puts that packet at the corresponding index in a fixed-size global byte chunk array. If the checksum indicates that the file is corrupted, the packet is ignored. In any other scenario, that is file is received without corruption, file receiver function sends an ACK message through the corresponding path to

the corresponding node for each and every packet it receives. When a packet that has a sequence and ACK number of 0 arrives to the file receiver, it deduces that the file transmission is over; it creates a file with the specified output file name, writes the data in the byte chunks to that file and terminates. In accordance with the case-specific implementation and the modified selective repeat method, the byte chunk array inside d resembles a bookshelf; in order to have the entire file, it is sufficient to put every book (payload) that arrives, into its corresponding place (index).

In our implementation, after that the file transfer is done, each script terminates and the script inside s prints the file transfer time to the "stdout". If one wants to execute this flow for just one file transfer; one should set margin of error threshold to -1 in s and repetition count to 1 in d. This way, one file transfer is completed reliably and all the scripts in the nodes terminate. Routing tables before and after link down operation in computers r1 and r2 are given in the Appendix with their related commands.

IV. Experiment Results

In this section we present our experiment results of experiment one and two in two subsections. We give corresponding graphs and tables and discuss the results. Figures *1* and *2* show the margin of error as 10% for each packet loss percentage since our experiments fall inside that region. However, as explained in the methodology part, we have stopped our experiments only after the margin of error dropped below our threshold 0.1. Thus, although the figures *1* and *2* show 10% for each, the real margins of error are reported in *Table 1* and *Table 2* in detail.

A. Experiment I

In this experiment we have transfered the file through $s - r3 - d$. We have emulated packet loss percentages of $5\%, 15\%$ and 28%. These emulations were done using the "tc/netem" command as explained in the methodology section.

The observation from the graph shows us that; as packet loss percentage increases, the file transfer time also increases. It is an expected behaviour from a reliable data transfer protocol since it will try to send the lost packets until it ensures the arrival of the packets. However this behaviour is observed to increase the overhead as the loss percentage grows.

B. Experiment II

In this experiment, as opposed to the first one, we send the file through two distinct paths, namely; $s - r1 - d$ and $s - r2 - d$. The experiments were conducted after the respective configurations were done on the network. Both of the mentioned distinct paths were up during the experiment.

The graph in *Fig. 2* shows a similar behaviour to the one in *Experiment I* and shows us the same direct relationship between packet loss percentage and file transfer time as expected.

C. Comparison

From the graph that compares the two experiments in *Fig. 3*, we can clearly infer that *Experiment II* is faster than *Experiment I*. The reason behind this is that the file division of multi-homing that uses 2 links to transfer separate parts of the file, utilises more links than that of *Experiment I* and thus it is faster. We can also observe that the difference between the two experiments in time is almost the same for every packet loss percentage, revealing the independency of multi-homing and packet loss.

V. Conclusion

This paper presents our own reliable data transfer protocol that we built on top of UDP. We used a modified version of selective repeat, we sent our files with pipelining method, we utilised more links by practicing multi-homing, we used single-ACKs and we implemented an auto-link-down detection of our own for reliability. We have tested the performance of our reliable data transfer protocol with two experiments. While the first experiment measured the performance of our reliable data transfer protocol through one link, the second experiment pointed out the effects of using multi-homing with two links. When we elaborate on the results of the experiments, as a common aspect, we see that in both experiments the increase in packet loss drastically increases the file transfer time, creating an overhead that negatively affects the performance. In addition to that, the difference between the two experiments reveal the fact that putting multi-homing to use increases our utilisation of network links and the overall performance by decreasing the file transfer time.

APPENDIX

```
beyazity@r1:~$ route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0         172.16.0.1      0.0.0.0         UG    1024   0        0 eth0
10.0.0.0        10.10.4.2       255.0.0.0       UG    0      0        0 eth2
10.10.0.0       10.10.1.1       255.255.252.0   UG    0      0        0 eth3
10.10.1.0       0.0.0.0         255.255.255.0   U     0      0        0 eth3
10.10.2.0       10.10.8.2       255.255.255.0   UG    0      0        0 eth1
10.10.2.2       10.10.1.1       255.255.255.254 UG    0      0        0 eth3
10.10.3.2       10.10.4.2       255.255.255.254 UG    0      0        0 eth2
10.10.4.0       0.0.0.0         255.255.255.0   U     0      0        0 eth2
10.10.4.0       10.10.8.2       255.255.254.0   UG    0      0        0 eth1
10.10.5.2       10.10.4.2       255.255.255.254 UG    0      0        0 eth2
10.10.6.0       10.10.8.2       255.255.255.0   UG    0      0        0 eth1
10.10.6.2       10.10.4.2       255.255.255.254 UG    0      0        0 eth2
10.10.8.0       0.0.0.0         255.255.255.0   U     0      0        0 eth1
172.16.0.0      0.0.0.0         255.240.0.0     U     0      0        0 eth0
172.16.0.1      0.0.0.0         255.255.255.255 UH    1024   0        0 eth0
beyazity@r1:~$ sudo ip link set dev $s_adapter down
beyazity@r1:~$ sudo ip link set dev $d_adapter down
beyazity@r1:~$ route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0         172.16.0.1      0.0.0.0         UG    1024   0        0 eth0
10.10.2.0       10.10.8.2       255.255.255.0   UG    0      0        0 eth1
10.10.4.0       10.10.8.2       255.255.254.0   UG    0      0        0 eth1
10.10.6.0       10.10.8.2       255.255.255.0   UG    0      0        0 eth1
10.10.8.0       0.0.0.0         255.255.255.0   U     0      0        0 eth1
172.16.0.0      0.0.0.0         255.240.0.0     U     0      0        0 eth0
172.16.0.1      0.0.0.0         255.255.255.255 UH    1024   0        0 eth0
beyazity@r2:~$ route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0         172.16.0.1      0.0.0.0         UG    1024   0        0 eth0
10.0.0.0        10.10.6.2       255.0.0.0       UG    0      0        0 eth1
10.10.0.0       10.10.2.2       255.255.252.0   UG    0      0        0 eth2
10.10.1.2       10.10.8.1       255.255.255.254 UG    0      0        0 eth4
10.10.2.0       0.0.0.0         255.255.255.0   U     0      0        0 eth2
10.10.2.0       10.10.6.2       255.255.254.0   UG    0      0        0 eth1
10.10.3.0       10.10.2.2       255.255.255.254 UG    0      0        0 eth2
10.10.4.0       10.10.8.1       255.255.255.254 UG    0      0        0 eth4
10.10.4.0       10.10.5.2       255.255.252.0   UG    0      0        0 eth3
10.10.5.0       0.0.0.0         255.255.255.0   U     0      0        0 eth3
10.10.6.0       0.0.0.0         255.255.255.0   U     0      0        0 eth1
10.10.6.0       10.10.6.2       255.255.254.0   UG    0      0        0 eth1
10.10.7.0       10.10.5.2       255.255.255.254 UG    0      0        0 eth3
10.10.8.0       0.0.0.0         255.255.255.0   U     0      0        0 eth4
172.16.0.0      0.0.0.0         255.240.0.0     U     0      0        0 eth0
172.16.0.1      0.0.0.0         255.255.255.255 UH    1024   0        0 eth0
beyazity@r2:~$ sudo ip link set dev $s_adapter down
beyazity@r2:~$ sudo ip link set dev $d_adapter down
beyazity@r2:~$ route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0         172.16.0.1      0.0.0.0         UG    1024   0        0 eth0
10.0.0.0        10.10.6.2       255.0.0.0       UG    0      0        0 eth1
10.10.1.2       10.10.8.1       255.255.255.254 UG    0      0        0 eth4
10.10.2.0       10.10.6.2       255.255.254.0   UG    0      0        0 eth1
10.10.4.0       10.10.8.1       255.255.255.254 UG    0      0        0 eth4
10.10.6.0       0.0.0.0         255.255.255.0   U     0      0        0 eth1
10.10.6.0       10.10.6.2       255.255.254.0   UG    0      0        0 eth1
10.10.8.0       0.0.0.0         255.255.255.0   U     0      0        0 eth4
172.16.0.0      0.0.0.0         255.240.0.0     U     0      0        0 eth0
172.16.0.1      0.0.0.0         255.255.255.255 UH    1024   0        0 eth0
```