Ceng 435
Term Project - Part 2 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 2, we developed UDP-based ”Reliable Data Transfer”, RDT, protocol of our own that supports pipelining and multi-homing for this homework.

For the first experiment we used the shortest path, that is found in the part 1 of the term project by using Dijkstra’s algorithm, of the given topology. We sent the given large file through this path by using our RDT protocol.

For the second experiment, we sent the same large file through the other paths which are r1 and r2 by dividing it into separate parts. We sent these parts of the message at the same time. Also, it is stated in the homework pdf that some of the links can be down so we tried to create a reliable implementation that updates the routing and makes a reliable transmission over the available link.

We simply described our design and implementation approach, our methodology, and our experimental results with their explanations and our comments to them.

We added description of how to run our scripts and how to do the experiments step by step in our README file.

II. Explanations

Before starting to the experiments, first we downloaded given topology and choose our resource as MAX InstaGENI on GENI portal. The given topology can be seen at the Figure 1.

However for the second part of the term project, we don’t have to use the links between r1-r2 and r2-r3.

To run the routers s, r1, r2, r3, an d we used the following lines on our terminal with the appropriate ports:

For node s: $ssh -i /.ssh/id_geni_ssh_rsa e2171882@pc2.instageni.maxgigapop.net -p 29610
For node r1: $ssh -i /.ssh/id_geni_ssh_rsa e2171882@pc2.instageni.maxgigapop.net -p 29611
For node r2: $ssh -i /.ssh/id_geni_ssh_rsa e2171882@pc2.instageni.maxgigapop.net -p 29612
For node r3: $ssh -i /.ssh/id_geni_ssh_rsa e2171882@pc2.instageni.maxgigapop.net -p 29613
For node d: $ssh -i /.ssh/id_geni_ssh_rsa e2171882@pc2.instageni.maxgigapop.net -p 29614

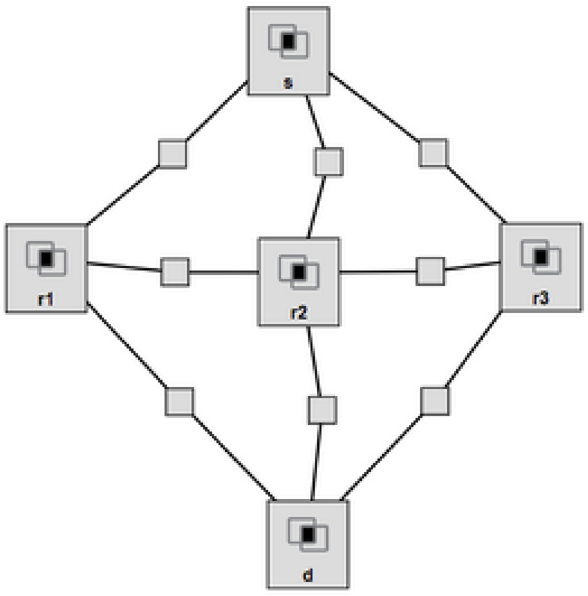


Fig. 1. Given topology

Also, we have to write both of the experiments on the same scripts. Therefore, we used a parameter to determine which experiment should run. These parameters are only added to run s.py and d.py files. To run the first experiment we used the following lines on terminal with appropriate ssh:

```
$ python s.py 1
$ python r3.py
$ python d.py 1
```

To run the second experiment, we used the following command lines on terminal with appropriate ssh:

```
$ python s.py 2
$ python r1.py
$ python r2.py
$ python d.py 2
```

If the sys.args[1] equals to 1 then the first experiment runs with the **s-r3-d** path, if sys.args[1] equals to 2 then the second experiment runs with the **s-r1-d** and **s-r2-d** paths.

Also, for each experiment we have different input and output files such as input1.txt, input2.txt, output1.txt, and output2.txt. To read the input files we used the following:

```
file1=open('input1.txt','r')
inp1=file1.read()

file2=open('input2.txt','r')
inp2=file2.read()
```

To write the output files, we used:

```
with open('output1/2.txt', 'w') as f:
    for item in DataRec:
        f.write("%s" % item)
```

A. Experiment 1

For experiment 1, we were expected to use the shortest path that is found by using Dijkstra's shortest path algorithm. We found the shortest path at the first part of this term project, the shortest path was **s-r3-d**. So, for this experiment we sent our large file which was exactly 5 MBytes through this path. We used UDP based RDT protocol of our own.

To send the given large file which is input1.txt, we first divide the file into chunks in the s.py script. Since our file is 5MB, we divided it into 5556 chunks whose length is 900. We stored these chunks in a buffer. With this method, we sent the file into the router r3. Also, we set our window size to 50. If the index of the buffer - currentWindow is less than and equal to 50, which is the window size, and i is less than 5556 and data is not sent before, this means that the data can be sent to the r3. To send the chunks we created a message that stores the data of the chunk, the checksum, and the index of the buffer. Between these part we added ”*̂_*”̂ signs to be able to separate these data, checksum, and index. Checksum helps to detect the bit errors. To calculate the checksum, we implemented a function as *chksum(data)*. This function takes a data and converts the characters into ASCII code and sums it. Then take its NOT. For this first experiment, we sent this created message to **r3** by using the following line: *clientSocket.sendto(message,('10.10.3.2',29613)).*

After sending the message to r3, s.py waits for ACK signal from r3 which means that the packet is sent without corruption. This method is used for reliable data transfer, RDT.

In r3.py script, r3 takes the message from s-r3 link by using the following lines:

```
server.bind(('10.10.3.2',portR3))
msg, addr = server.recvfrom(2048)
```

This server binds s to r3 link with the r3's port. After receiving the message r3 sends this into d by using the following part:

```
clientSocket = socket(AF_INET, SOCK_DGRAM)
clientSocket.sendto(msg,('10.10.7.1', dPort))
```

This clientSocket, UDP socket, sends the message by using link between r3 and d to port of d. After sending the message to route d, r3 waits for the ACK signal that is sent by d. This ACK signal is sent when the message is nor corrupted.

In d.py script, for the experiment 1 we created a function named d_r3() which takes the message from r3 and checks whether the message is corrupted or not. To check the message, we divided the received message into parts by determining where the ”*̂_*”̂ signs are. To find these signs we used message.find(”*̂_*”̂). After finding the index of these signs, we received the data, checksum, and index of the buffer that we added together in s.py script. Then by using sum(data) function which takes data, converts it in to ASCII code and sums it up, we calculated the sum of the data. If the sum of the data + checksum equals to -1, the data is received without corruption; therefore, we sent ACK signal to r3. This ACK signal indicates the index of the data that is not corrupted. Also, we created a file named output1.txt and write the received data into this file. This output file's size is the same with the input file's size, and our algorithm send the exact file to the router d from router s.

r3.py script takes the ACK signal sent from d.py, and sends it to s.py. If the ACK signal is received, the message is sent without corruption, else we resent the message. When the window has received ACK signals for all of its chunks, we updated the currentWindow to the index of the next chunk that has not received ACK signal.

To time-out in s.py script, we implemented a function named nextWindow() which sleeps for 1.5 seconds and then changes the index of the buffer into the currentWindow. This approach is similar to rdt3.0, we waited for 1.5 seconds for ACK signal, and if it is not received in this time we resent the message again.

To support pipelining and multi-homing, we used threads in our project. We implemented different functions for sending message, timeout, and receiving ACK signals. Then we run these functions with different threads to use these functions at the same time.

B. Experiment 2

For the experiment 2 part, we have to send the input message through paths **s-r1-d** and **s-r2-d**. To separate the message into two parts we used the following algorithm. We created two different servers at s.py script and bind those with r1 and r2 with the following codes:

```
serverSocketR1.bind(('10.10.1.1',29614))#ip of r1
serverSocketR2.bind(('10.10.2.2',29614))#ip of r2
```

To serverSocketR1 we bind it with the link between s to r1 and port of s, to serverSocketR2 we bind it with the link between s to r2 and port of s.

We first created two different functions as senderR1() and senderR2(). To send the data, we created the message as explained in the section *A.Experiment 1*. We added data, checksum which is found by using chksum(data) function, and index of the buffer, i. Again to separate these parts, we added ”*̂_*”̂ sign. A the d.py script, we separate these parts by looking into these signs. If the buffer index of the data is even which means i%2==0, then we sent this message to router r2. If the buffer index is odd which means i%2==1 then we sent this packet to router r1. To apply multihoming we used threads. We run each of the functions as different threads so that they can run at the same time. With this method we sent the data from both of the paths at the same time.

Scripts r1.py and r2.py, takes the given message from node s and sends it to node d. And it also receives ACK signal from node d and sends it to node s. Scripts r1.py and r2.py takes the message from s.py and send it to the node d by using the following lines:

```
msg, addr = server.recvfrom(2048)
clientSocket = socket(AF_INET, SOCK_DGRAM)
clientSocket.sendto(msg,(IP, dPort))
clientSocket.close()
```

The IP value is 10.10.5.2 for r2.py and IP is 10.10.4.2 for r1.py. dPort value is the port of the node d. After sending the message to node d, it waits for ACK signal from node d.

In script d.py, we took the message from nodes r1 and r2. We created two different functions to take the messages from r1 and r2. To create the whole data, we separate the message into the parts that we created in script s.py, data, checksum, and index of the buffer. By looking into these buffer index we created the whole data again. When we took the message we control it by checking whether sum(data)+checksum, as explained in the section *A.Experiment 1*, equals to -1 or not. If it is -1 then we got the message without corruption so we sent ACK signal to the appropriate node. To send the ACK signal, we used the following line:

clientSocketAck.sendto(str(fake_i),('10.10.7.2',r3Port))

This line sends the index of the buffer, fake_i, to r1/2.py and it sends it to the script s.py.

In scripy d.py, if the message is received without corruption it writes it into the output2.txt file. We compared the output file with the input file, and they have the same size which is 5MB and they are exactly the same. So this means that the data is sent by RDT, reliable data transfer.

The ACK signal is received by s.py script, if the ACK signal is received this means that the message is sent without corruption. If signal is not received then it resents the message again by looking to the sent buffer index to prevent packet loss and provide reliable data transfer.

Also, in homework file it is stated that some of the links, s-r1 and s-r2, can be down. We first find the eth? for each link between nodes by using the following:

ip route get IP

where IP is the ip of the link. Then by using the eth? we get, we run the following command:

ip link set dev -interface- down

where -interface- is the eth? that we found by using the command above.

To deal with this problem we used time-out at script s.py. We sleep for 1.5 seconds and if it is still not received from the node this means that the link is down, we sent the message with the other node. With this method, even when one of the links is down, the messages can be sent to node d without any corruption.

C. Terminating Scripts

For this part of the term project we should terminate from d.py and s.py scripts when all of the input is sent. When we tried to terminate from d.py and s.py scripts, we faced with a problem. So, we created a new list with zeros whose size is 5556, which is the same with the number of chunks. Then when we received the data at some index, we changed that index at DataSended list into 1. So at every function we checked whether the DataSended has 0 or not. If the list don't have any zeros on it this means that every message is sent. So we brake from while true loop when list has no zeros. This is shown in the following lines:

```
while True:
    ....
    if 0 not in DataSended:
            flag=1
            break
```

We added these if condition to every function on s.py script, so that the while loop breaks when all of the data is sent. We also change flag, which was initially zero, to 1. So we have three flags for each nodes r1,r2, and r3. Then we checked these flags at the senderR?() functions because at these functions the DataSended list has zero when these functions are running. If the flag is 1 then we break from the while loop because all of the data is sent already.

D. Multihoming

To provide multihoming, we used threads in our implementation. Threads helps to run the functions at the same time. We imported threads with the following line:

from threading import Thread

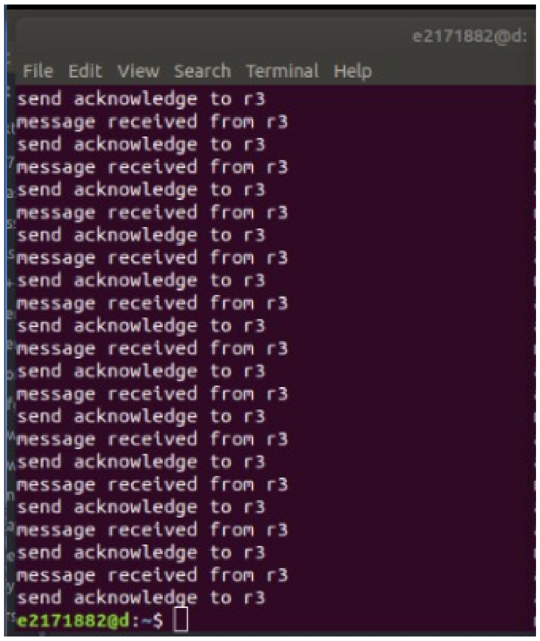
Then we created and joined threads by using the following lines:

```
sender1 = Thread(target = senderR1,args=())
sender1.start()
sender1.join()
```

So that in the experiment 2, the message is sent separately through paths **s-r1-d** and **s-r2-d** simultaneously. The terminal screenshots for the experiment 1 can be seen below. As you can see from the screenshots Figure 2, 3, and 4 above, node s sends the data to r3 which sends it to d. Then d sends ACK when the data is received without corruption to r3 and r3 sends ACK signal to node s. If ACK signal for a specific index then node s resends the message with that specific index.

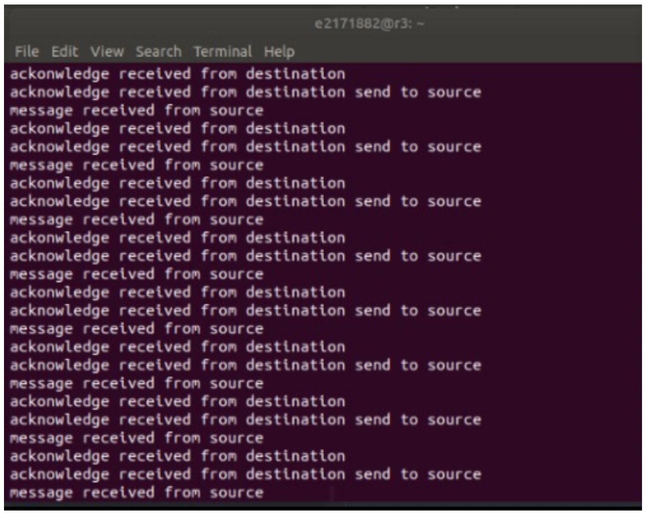
The terminal screenshots for the experiment 2 can be seen below.

As you can see from the screenshots Figure 5,6, and 7, paths **s-r1-d** and **s-r2-d** are running at the same time. So the experiment 2 is faster than the experiment 1 because experiment 2 sends the message as two parts through two



```
send acknowledge to r3
message received from r3
send acknowledge to r3
message received from r3
send acknowledge to r3
message received from r3
send acknowledge to r3
message received from r3
send acknowledge to r3
message received from r3
send acknowledge to r3
message received from r3
send acknowledge to r3
message received from r3
send acknowledge to r3
message received from r3
send acknowledge to r3
message received from r3
send acknowledge to r3
message received from r3
send acknowledge to r3
message received from r3
send acknowledge to r3
e2171882@d:~$
```

Fig. 2. Terminal of d



```
ackonwledge received from destination
acknowledge received from destination send to source
message received from source
ackonwledge received from destination
acknowledge received from destination send to source
message received from source
ackonwledge received from destination
acknowledge received from destination send to source
message received from source
ackonwledge received from destination
acknowledge received from destination send to source
message received from source
ackonwledge received from destination
acknowledge received from destination send to source
message received from source
ackonwledge received from destination
acknowledge received from destination send to source
message received from source
ackonwledge received from destination
acknowledge received from destination send to source
message received from source
ackonwledge received from destination
acknowledge received from destination send to source
message received from source
```

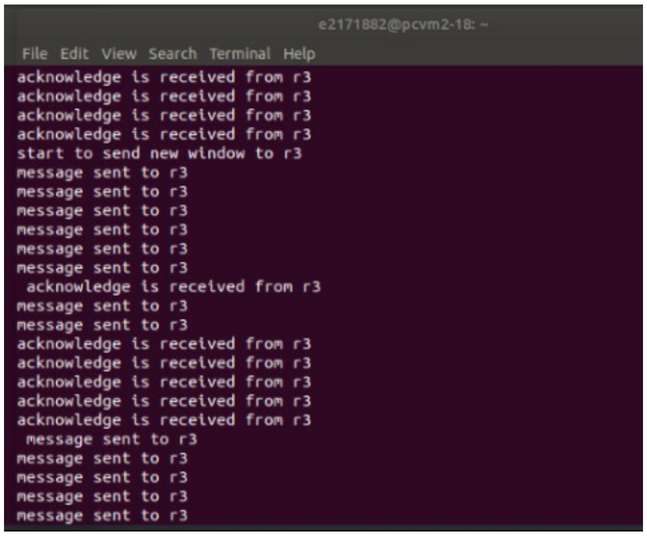
Fig. 3. Terminal of r3

different paths while experiment 1 sends it through only one path, which is the shortest path by Dijkstra's algorithm **s-r3-d**. Sending message separately by two paths at the same time is more faster than sending by shortest path only.

E. Experimental Results

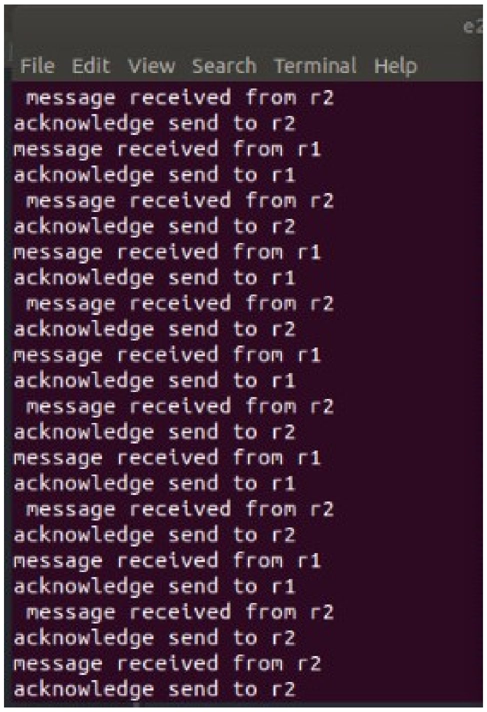
For the experimental part of the project we used the following command lines for all of the nodes in the topology and for both of the experiments:

$sudo tc qdisc add dev [INTERFACE] root netem loss 5% delay 3ms
$sudo tc qdisc change dev [INTERFACE] root netem loss 15% delay 3ms



```
acknowledge is received from r3
acknowledge is received from r3
acknowledge is received from r3
acknowledge is received from r3
start to send new window to r3
message sent to r3
message sent to r3
message sent to r3
message sent to r3
message sent to r3
message sent to r3
 acknowledge is received from r3
message sent to r3
message sent to r3
acknowledge is received from r3
acknowledge is received from r3
acknowledge is received from r3
acknowledge is received from r3
acknowledge is received from r3
 message sent to r3
message sent to r3
message sent to r3
message sent to r3
message sent to r3
```

Fig. 4. Terminal of s



```
 message received from r2
acknowledge send to r2
message received from r1
acknowledge send to r1
 message received from r2
acknowledge send to r2
message received from r1
acknowledge send to r1
 message received from r2
acknowledge send to r2
message received from r1
acknowledge send to r1
 message received from r2
acknowledge send to r2
message received from r1
acknowledge send to r1
 message received from r2
acknowledge send to r2
message received from r1
acknowledge send to r1
 message received from r2
acknowledge send to r2
message received from r2
acknowledge send to r2
```

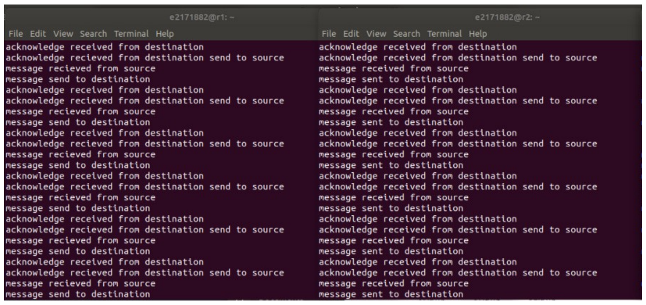
Fig. 5. Terminal of d

$sudo tc qdisc change dev [INTERFACE] root netem loss 38% delay 3ms

To find [INTERFACE] part we again used the following command, where IP is the ip of the link between selected nodes:

$ip route get IP

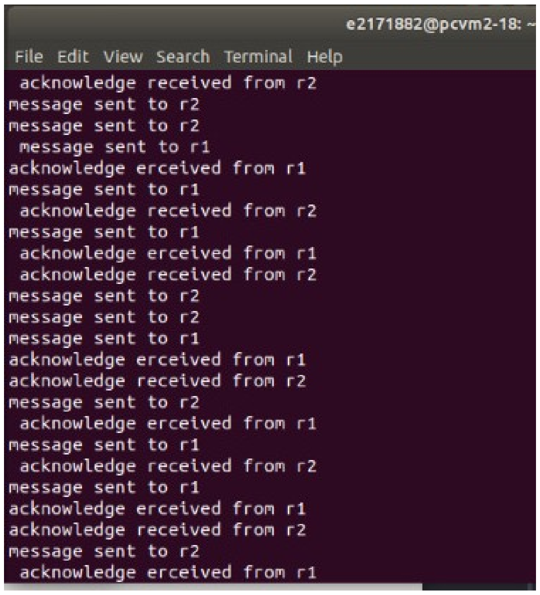
For the first line we used "add" instead of "change" because when we first tried with the "change" it gave the following error:


e2171882@r1: ~
File Edit View Search Terminal Help
e2171882@r2: ~
File Edit View Search Terminal Help

```
acknowledge received from destination
acknowledge recieved from destination send to source
message recieved from source
message send to destination
acknowledge received from destination
acknowledge recieved from destination send to source
message recieved from source
message send to destination
acknowledge received from destination
acknowledge recieved from destination send to source
message recieved from source
message send to destination
acknowledge received from destination
acknowledge recieved from destination send to source
message recieved from source
message send to destination
acknowledge received from destination
acknowledge recieved from destination send to source
message recieved from source
message send to destination
acknowledge received from destination
acknowledge recieved from destination send to source
message recieved from source
message send to destination
```

```
acknowledge received from destination
acknowledge received from destination send to source
message received from source
message sent to destination
acknowledge received from destination
acknowledge received from destination send to source
message received from source
message sent to destination
acknowledge received from destination
acknowledge received from destination send to source
message received from source
message sent to destination
acknowledge received from destination
acknowledge received from destination send to source
message received from source
message sent to destination
acknowledge received from destination
acknowledge received from destination send to source
message received from source
message sent to destination
acknowledge received from destination
acknowledge received from destination send to source
message received from source
message sent to destination
```

Fig. 6. Terminal of r1 and r2



```
 acknowledge received from r2
message sent to r2
message sent to r2
 message sent to r1
acknowledge erceived from r1
message sent to r1
 acknowledge received from r2
message sent to r1
 acknowledge erceived from r1
 acknowledge received from r2
message sent to r2
message sent to r2
message sent to r1
acknowledge erceived from r1
acknowledge received from r2
message sent to r2
 acknowledge erceived from r1
message sent to r1
 acknowledge received from r2
message sent to r1
acknowledge erceived from r1
acknowledge received from r2
message sent to r2
 acknowledge erceived from r1
```

Fig. 7. Terminal of s

RTNETLINK answers: File not exist

To overcome this problem, we used ”add” first.

We added the following line to the top of the d.py file,

```
start_time = time.time()
```

and also the following line when the ackRec() function receives the ACK signal from the r1, r2, or r3 node:

```
end_time = time.time()
transfer_time = end_time-start_time
```

When we plot the graph based on the file transmission time and loss percentage. The graphs x-axis shows the packet loss percentage (5%, 15%, 38%) and y-axis shows file transfer time. We obtained the following graph, Figure 8 for the experiment 1: File transfer time is increased when the packet loss percentage is increased. Because when there is loss of the packet, node s sends that lost message again and it causes transfer time to increase. So the result became like we expected. We used the following Matlab code to plot the Figure 8:

```
x=[5,15,38];
y=[285.12893, 520.1300, 1397.64794];
```

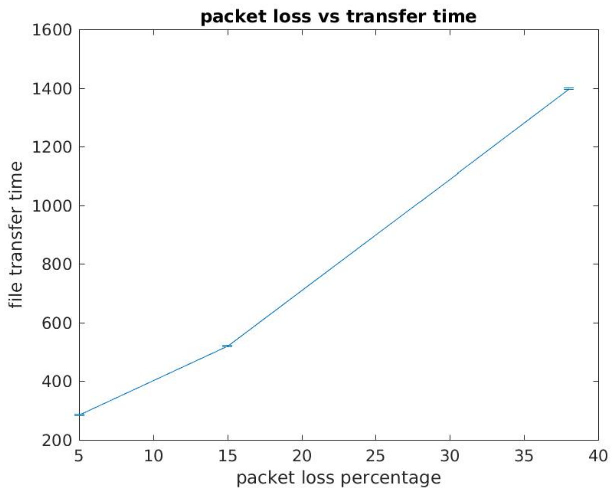


Fig. 8. The Packet Loss Percentage vs File Transfer Time

```
e=[2.4,2.4,2.4];
errorbar(x,y,e);
xlabel('packet loss percentage');
ylabel('file transfer time');
title('packet loss vs transfer time');
```

The graphs x-axis shows the packet loss percentage (5%, 15%, 38%) and y-axis shows file transfer time. We obtained the following graph, Figure 9 for the experiment 2: Again in this graph file transfer time is increased when packet loss is increased. Still the experiment 2’s file transfer time is smaller than the experiment 1’s transfer time because experiment 2 uses two different path while experiment 1 uses only one path. As you can see the increase rate between packet loss 15% and 38% is higher than increase rate between packet loss 5% and 15%. We used the following Matlab code to plot the Figure 9:

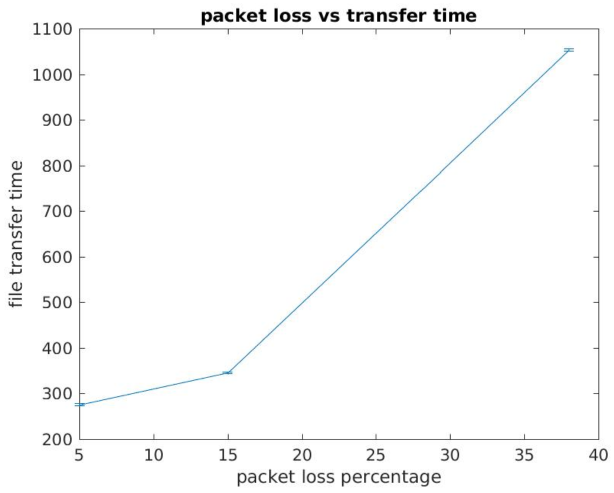


Fig. 9. The Packet Loss Percentage vs File Transfer Time

```
x=[5,15,38];
```

```
y=[274.85041, 345.144378, 1053.5478];
e=[2.4 ,2.4 ,2.4];
errorbar(x,y,e);
xlabel('packet loss percentage');
ylabel('file transfer time');
title('packet loss vs transfer time');
```

F. Workshare

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

III. Conclusion

This part of the term project has two different parts for experiment, and also another part for calculating experimental results. For the experiment 1, we used the shortest path that we found on the part1 of this project by Dijkstra's algorithm, this path was **s-r3-d** and we sent our file from this path with our RDT protocol.

For the experiment 2, we used the other paths which are **s-r2-d** and **s-r3-d** at the same time. We separate the input file into two and sent it through these paths. To provide multihoming we used threads.

To calculate the experimental result, we found the file transfer time, and plotted the loss percentage vs file transfer time graph. As we expected, the file transmission time of experiment 2 is smaller than the experiment 1. Also when the packet loss is increased, the file transfer time is increased because node s sends the message again to prevent packet loss due to the RDT protocol.

References

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