CENG435 Group 25 TP Part-2 Report

Ali Alper Yüksel, Student ID: 2036390
Yunus Emre Gürses, Student ID: 2099083

1. Processes of Part-2

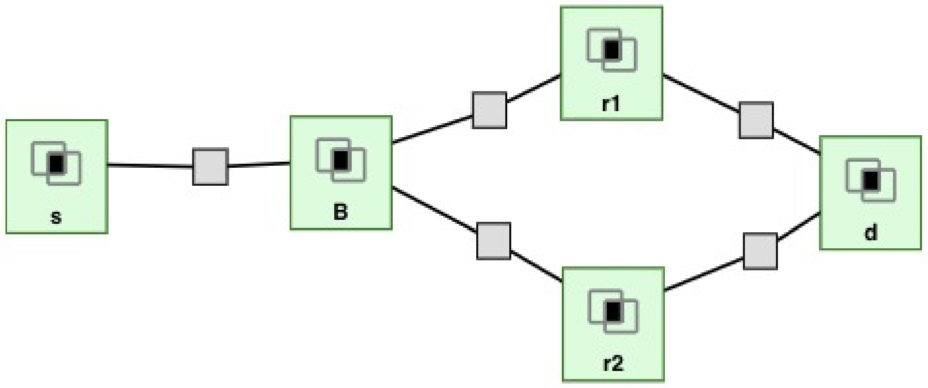
In this part-2 of project, we followed these steps:

1. Using Geni platform with the experience from part-1
2. Understanding the topology and implementing our network system depending on this topology
3. Using Python socket programming to build system
4. Implementing RDT over UDP
5. Uploading codes into virtual machines
6. Testing communication
7. Using netem commands to test packet loss, corruption and reordering and their relationships with file transfer time.

2 .Geni Platform

With the experiences about how to use Geni platform from part-1, we built a topology with xml file uploaded to odtuclass and managed virtual machine in this topology via SSH.

3.Topology and Programming Part



As it can be seen in topology, we have five hosts. s stands for source node, d stands for destination node, r1 and r2 stands for router nodes and B is considered as broker. So, explanations of components of this topology are following:
1. s (Source Node): Acts as source device which collects data from input file. It just sends data to broker (B) with TCP.
2. B : Acts as broker. It listens s and receive data when it is sent from source with TCP and packetize it. Then, send these packets to r1 and r2 with UDP. Also listens routers and receive control packets when they are sent from routers.
3. r1 and r2 (Router Nodes): They listen b and take packets when it is sent from broker and send these packets to d with UDP. Also listen d and take control packets when they are sent from destination.
4. d (Destination Node): It listens r1 and r2 and when packets are sent, d receives these packets. Also can create control packets and send to routers.

The tables below show routing tables for s,b,r1, r2 and d. We can understand where is the next station and destination to send for reaching our packet to one from these nodes.

Next node	**Destination**
B	B
B	r1
B	r2
B	d

Table 1: *s* Route Table

Next node	**Destination**
s	s
r1	r1
r2	r2
r1 or r2	d

Table 2: *B* Route Table

Next node	**Destination**
B	s
B	B
d	d

Table 3: *r*1 Route Table

Next node	Destination
B	s
B	B
d	d

Table 4: $r2$ Route Table

Next node	Destination
r1 or r2	s
r1 or r2	B
r1	r1
r2	r2

Table 5: d Route Table

Since, there is no restriction about choice of programming language which we use, we used Python for this assignment in programming part. Python has more useful modules and its syntax needs less effort than other programming languages.

In source side, we create TCP connection to send input file namely input.txt. To create connection, we use TCP socket programming. We imported "socket" library. To calculate file transfer time and work on time data, one Python time module is used which called as datetime..

For broker side, first we connect to source with TCP socket programming to take data from source and write it into input2.txt file. When we create socket object, second parameter takes `SOCK_STREAM` constant to represent TCP. This file is useful to detect whether all data are sent from source to broker or not.

For broker and destination sides, we create UDP socket protocol to connect routers by using UDP socket programming. When we create socket object, second parameter takes `SOCK_DGRAM` constant to represent UDP. Then, we implement RDT over UDP. When we implement, in order to provide pipelining, we use go-back-N approach. In broker side, we make a list with sequence number, data and checksum of data. We create a new function to detect the checksum of data namely `ip_checksum`. Then, we serialize this list with "pickle.dumps()" function from "pickle" library which is imported in script file. Then, our lists are ready to send to routers as packets.

In destination side, we take packets which are sent from routers with "pickle.loads()" function. First, we check the value of checksum received against checksum calculated to determine whether the packet is corrupted or not. If not corrupted, we again check the value of sequence number received against value of expected sequence number to determine whether packet is received in order or not. If received in order, we write data into again a new txt file namely output.txt to check whether all data are sent from routers to destination or not. Then, we make a list to create control packets with expected sequence number and checksum of expected sequence number to send to routers. If received sequence number is not same as expected sequnce number, we discard packet and resend ACK for most recently received inorder packet.

When control packets are sent to broker side via routers, we check value of checksum received against checksum calculated to detect whether the packet is corrupted or not. If values are same, we slide window and reset timer.

At broker side, we saved the time of last ACK received. While we send packets, we check the difference between current time and last ACK received time. If their difference is greater than spesific amount of time, we send all the packets again in the window. Similiar mechanism is applied in destination side also. However, the difference is that we check the difference of time if ”endoffile”.

To connect broker with destination without any script, we use route commands.
For r1:

```
sudo route add -host 10.10.2.1 gw 10.10.2.2
sudo route add -host 10.10.3.2 gw 10.10.2.2
```

4. Uploading codes into virtual machines

After writing codes into three seperate files, we uploaded files into virtual machines by using SSH. Also we used VIM editor to edit these python script files. Also with scp command, we can upload input file into SSH with this command:

```
scp -P [port number] /path/to/file e2099083@pc1.lan.sdn.uky.edu:
```

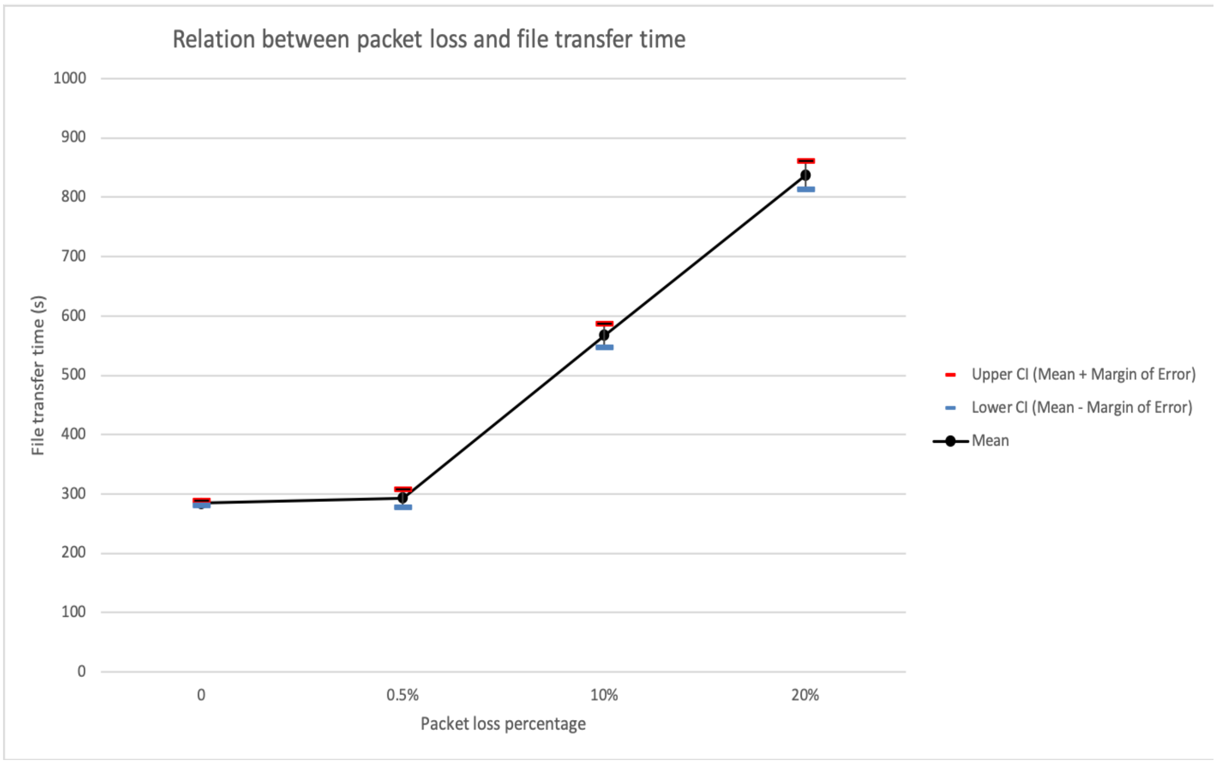
5. Testing packet loss, corruption and reordering and their relationships with file transfer time

We used netem commands to simulate and build our graphs about relation between packet loss, corruption and reordering and file transfer time. Also to generate more realistic results, we run our system 10 times. In our graphs, we use %95 confidence interval.

To obtain packet loss with %0.5, %10 and %20, we added following command into virtual machines:

```
tc qdisc change dev [INTERFACE] root netem loss 0.5% corrupt 0% duplicate
0% delay 3 ms reorder 0% 0%
tc qdisc change dev [INTERFACE] root netem loss 10% corrupt 0% duplicate
0% delay 3 ms reorder 0% 0%
tc qdisc change dev [INTERFACE] root netem loss 20% corrupt 0% duplicate
0% delay 3 ms reorder 0% 0%
```

For broker node, we applied eth3 device to use link between broker and r1.
For r1 node, we applied eth1 device to use link between destination and r1.
For destination node, we applied eth2 device to use link between destination and r1.



For no packet loss:
Mean = 284
Standart deviation = 5,391351098
Confidence = 3,856740241
Upper CI = 288,0567402
Lower CI = 280,3432598

For %0.5 packet loss:
Mean = 293
Standart deviation = 21,62817093
Confidence = 15,47186144
Upper CI = 308,4718614
Lower CI = 277,5281386

For %10 packet loss:
Mean = 567
Standart deviation =28,28427125
Confidence = 20,23334877
Upper CI = 587,2333488

Lower CI = 546,7666512

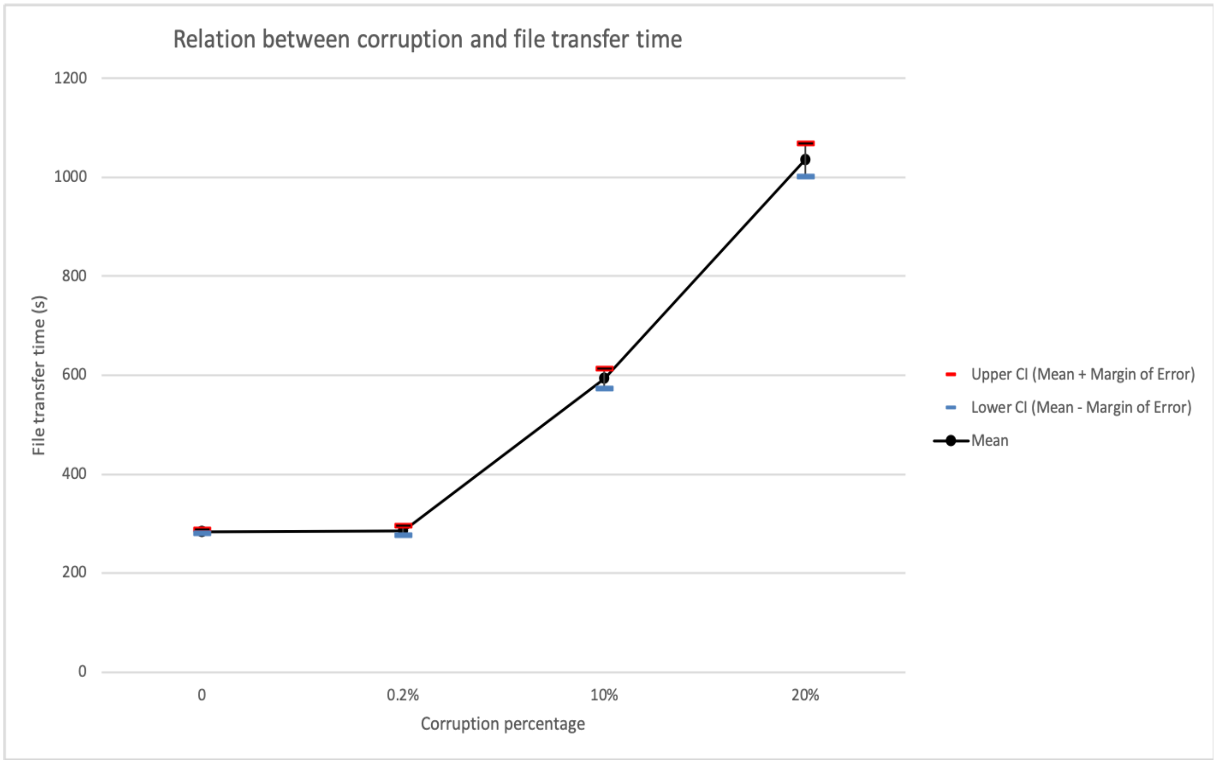
For %20 packet loss:
Mean = 837
Standart deviation = 32,99831646
Confidence = 23,60557356
Upper CI = 860,6055736
Lower CI = 813,3944264

When we apply packet loss to devices, we can easily notice that file transfer time is also increased. We use UDP in each link, and contrary to TCP, UDP provide no recovery for loss packets.

To obtain corruption with %0.2, %10 and %20, we added following command into virtual machines:

```
tc qdisc change dev [INTERFACE] root netem loss 0% corrupt 0.2% duplicate
0% delay 3 ms reorder 0% 0%
tc qdisc change dev [INTERFACE] root netem loss 0% corrupt 10% duplicate
0% delay 3 ms reorder 0% 0%
tc qdisc change dev [INTERFACE] root netem loss 0% corrupt 20% duplicate
0% delay 3 ms reorder 0% 0%
```

For broker node, we applied eth3 device to use link between broker and r1.
For r1 node, we applied eth1 device to use link between destination and r1.
For destination node, we applied eth2 device to use link between destination and r1.



For no corruption:
Mean = 284
Standart deviation = 5,391351098
Confidence = 3,856740241
Upper CI = 288,0567402
Lower CI = 280,3432598

For %0.2 corruption:
Mean = 286
Standart deviation = 13,3666251
Confidence = 9,561907578
Upper CI = 295,5619076
Lower CI = 276,4380924

For %10 corruption:
Mean = 594
Standart deviation =28,28427125
Confidence = 20,23334877
Upper CI = 614,2333488
Lower CI = 573,7666512

For %20 corruption:
Mean = 1036

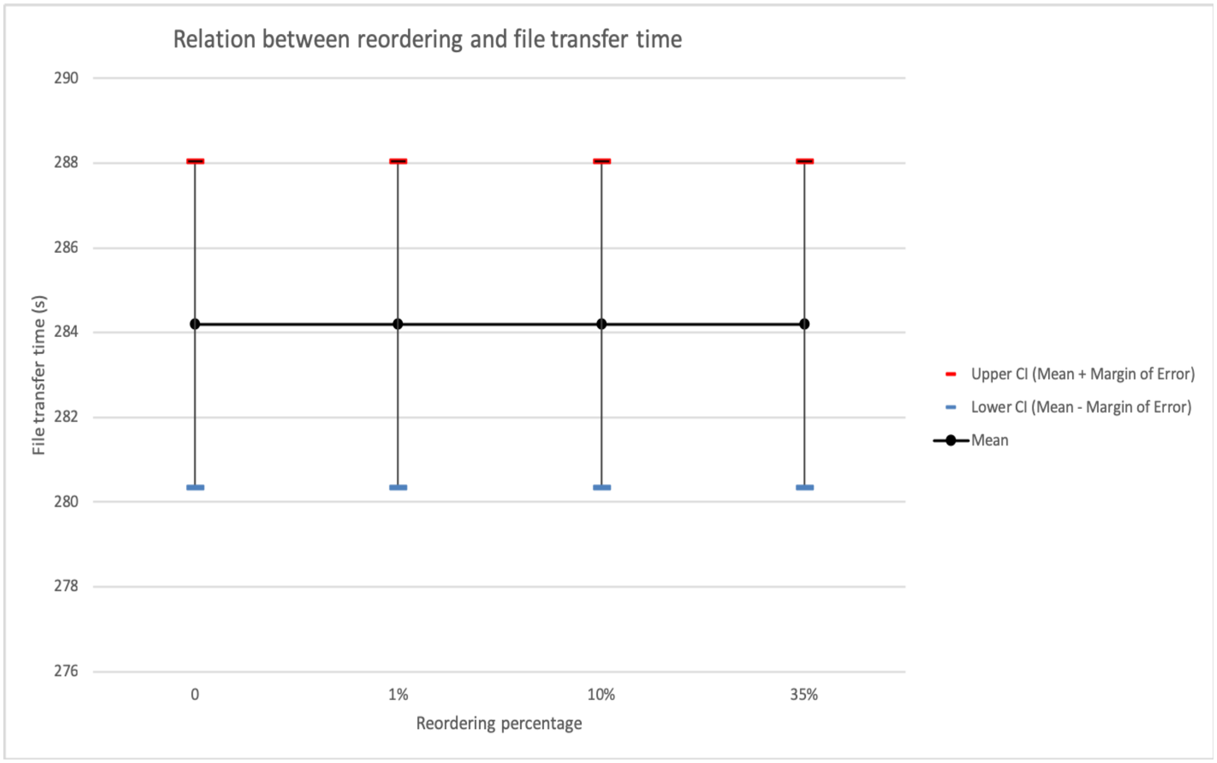
Standart deviation = 47,14045208
Confidence =33,72224795
Upper CI = 1069,722248
Lower CI = 1002,277752

When we apply corruption to devices, we can easily notice that file transfer time is also increased. If the received packet is corrupted in receiver, then receiver resend the ACK for most recently received inorder packet.

To obtain reordering with %1, %10 and %35, we added following command into virtual machines:

```
tc qdisc change dev [INTERFACE] root netem loss 0% corrupt 0% duplicate 0%
delay 3 ms reorder 1% 50%
tc qdisc change dev [INTERFACE] root netem loss 0% corrupt 0% duplicate 0%
delay 3 ms reorder 10% 50%
tc qdisc change dev [INTERFACE] root netem loss 0% corrupt 0% duplicate 0%
delay 3 ms reorder 35% 50%
```

For broker node, we applied eth3 device to use link between broker and r1.
For r1 node, we applied eth1 device to use link between destination and r1.
For destination node, we applied eth2 device to use link between destination and r1.



For no reordering:

Mean = 284
Standart deviation =5,391351098
Confidence = 3,856740241
Upper CI = 288,0567402
Lower CI =280,3432598

For %1 reordering:
Mean = 284
Standart deviation =5,391351645
Confidence = 3,856740542
Upper CI = 288,0567432
Lower CI =280,3432123

For %10 reordering:
Mean = 284
Standart deviation =5,391351122
Confidence = 3,8567406474
Upper CI = 288,05674453
Lower CI =280,3432598

For %35 reordering:
Mean = 284
Standart deviation =5,391351234
Confidence = 3,85674054231
Upper CI = 288,05674123
Lower CI =280,34325143

When we apply reordering to devices, we can easily notice that file transfer time has no change. The reason of that is in UDP, the packet with sequence number smaller than received sequence number can arrive is if there was reordering in the network and the receiver gets an old packet; for such packets, the receiver can safely not send an ACK because it knows that the sender knows about the receipt of the packet and has sent subsequent packets.