Middle East Technical University Department of Computer Engineering

CENG 435

DATA COMMUNICATIONS AND NETWORKING

Fall 2023-2024

Socket Programming Assignment

Name Surname: Alkım Doğan, Koray Özgür
Student ID: 2521482, 2580843

Coding Process

TCP

We first started with TCP part. This part was obviously easier than UDP part since we did not need any additional feature. We made use of object oriented programming in this assignment. There are two different files for each docker, one for server and one for client. Main python3 files only contain calls of constructors and functions calls that help us receive and send the data. There are 3 functions for TCP client, which are one for receiving large file, one for receiving small file and one for closing the socket. As for TCP server, there are 2 functions. One of them is for sending the data, whereas the other is for closing the socket. One of the trick we used was the fact that we sent the length of the file before we sending the actual file. The purpose was the fact that receiver was supposed to know the length of the payload/data in order to fill the parameter of the socket.recvfrom(length) function. Another trick was we handled the IP addresses dynamically for the sake of easiness. We just write IP address of the server to a txt file, after that we read this file from the client and receive the packets. Because of this trick, we did not have to change the IP address each time we change our computer. The last thing to mention is constants.py file. This file contains some generic numbers we use for transferring the large and small objects.

UDP

As for UDP part, this part was much challenging compared to TCP part. We again have main python3 files that actually the same purpose as the main files in TCP part. One small different is that we have array of buffers, each buffer representing another file. One of the most important part is about packet.py.we created a packet class, since RDT requires some additional header files such as sequence number, checksum and stream id (file id) and so on. We assigned minimum number of bytes for each field of the header. Packet.py files also contains pack and unpack functions. These functions use struct library for packing and unpacking, and MD5 checksum functions from hashlib library. There is again a constants.py file that contains some variable length and sizes. We should indicate that for each experiment we had to change some length and sizes since the results are highly depended on the length of payload, size of the window and so on.

UDP server implements some reliable data transfer functions. UDP server has a data attribute which is yielded from the main to our server class. This data attribute is a list of compressed payloads that we

read from large and small objects. The idea of compressing the payloads was advised by Ertan Hoca to us. We tested and observed compression makes the system run a little bit faster. If the file is ended, we send empty payload for the indication of the end of the file. We have a window which is a deque. We start by filling the window for corresponding sequence numbers. Afterwards, we start sending the packets that are not previously sent. Before we send packet, the header fields are filled. We fill the timestamp header and mark the packet as sent because we need to know which packets are sent or not for retransmission. We apply selective repeat with cumulative ACK. If the ACK packet that is sent by receiver corresponds to a packet in the window, we pop the packets with smaller sequence number since this means the receiver has got the packets up to that specific ACK sequence number. Otherwise, we use duplicate ACK counter. We count up to three for detecting duplicate ACK case. Then, we send the first packet in the window since duplicate ACK detected. Finally, we should mention retransmission case. If the receiver has received the packet and the ACK packet that the receiver has got lost, then we should detect it by using timeout. We check if the timeout has occured for each packet that are marked as sent in the window. After detecting timeout, we update the timestamp header field and resend the packet.

UDP client has a window like server as well as a boolean array that checks whether the transmission for that file is done or not. We need that because the order of the packets may change because of loss case. If that happens,the order of the file transmission may change as well. If we receive and empty payload, we mark that file as done in that boolean array. That is the reason why we need a boolean array. We again start by filling the window with corresponding ACK packets which is very similar to UDP server. Then, we wait for packet to receive. If the received packet corresponds to a ACK packet in the current window, we mark that packet as RECEIVED. Otherwise, which is if the received packet is not in the ACK window, we basically ignore it (because cumulative/duplicate ACK handles). Finally, We pop and yield the first element of the deque until the front of the queue is not received, and fill the window after this process. This helps us to send both cumulative and duplicate ACK. Since we compressed the payload with zlib, we decompress the payload before we yield it to the caller in the main python3 file.

How did we plot the figures?

For each experiment, we run the following command for calculating the time.

```
time python3 <python_script.py>
```

This command gives us the calculated time. We run this command for 30 times for each experiment. Each time we deleted the netem rules This was how we recorded the data.

We also coded a basic python3 script for plotting the figures. We made use of numpy, matplotlib, seaborn and scipy libraries. Each figures contains corresponding means and confidence intervals. Means of the plots are shown with red dotted line, whereas confidence interval is shown as black thick line. The x-axis for the plots corresponds to experiment trial. Each plots have 30 experiment trials, from 0 to 29 both included. As of the other axis, the y-axis contains the data we gathered from our each experiment. This corresponds to running time of the socket.

For the python script, you can click here.

Note: While running UDP, we first run client before running the server. We observed this is a little bit faster than the reverse order.

Experiment 1- No Ethem Rules

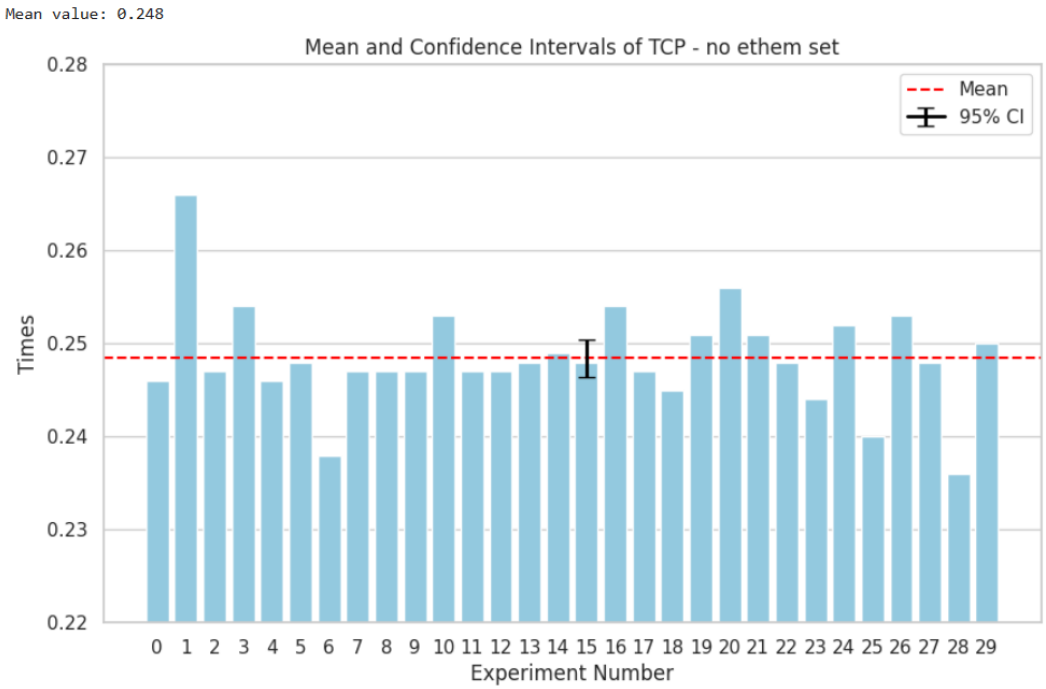

Figure 1: TCP plot (Benchmark, no tc/netem rules applied.)

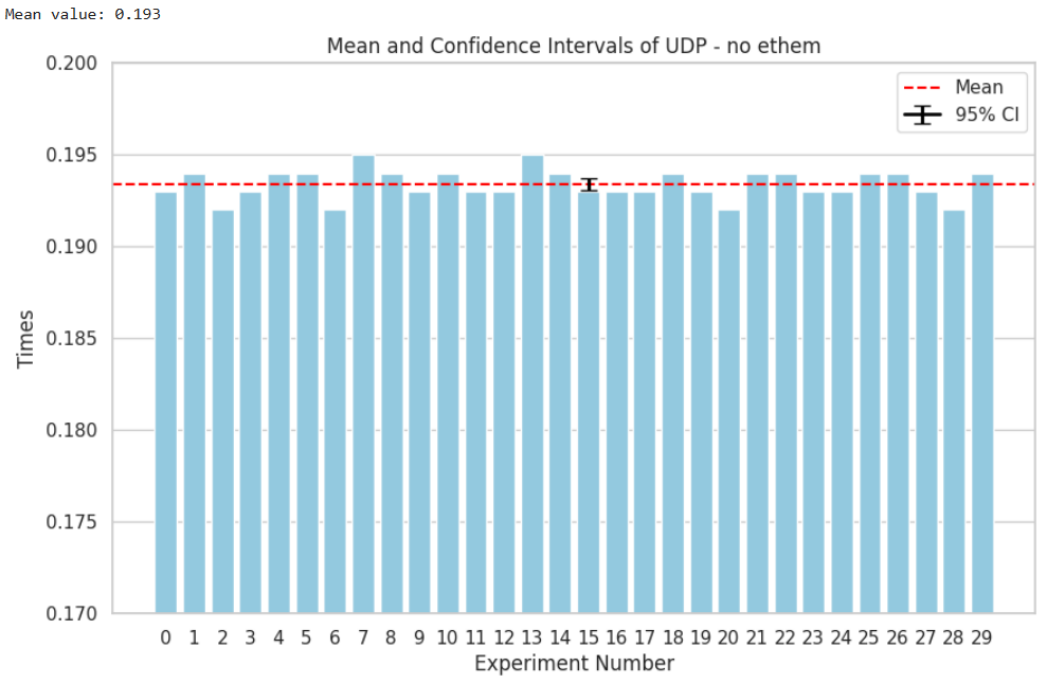

Figure 2: UDP plot (Benchmark, no tc/netem rules applied.)

Analyze For this experiment, UDP used 10000 as a payload size and 21 as a window size. Here, we find out that 10000 is the optimal payload size. We think it is a big number for a payload since it sends the small object without segmentation but this is how UDP is faster than TCP without netem rules applied.

The first figure contains the TCP graph, whereas the second figure contains UDP graph. Both experiments are done with no ethem rules set. We can see the mean of the TCP is equal to 0.248, while the mean of the UDP is equal to 0.193 both measured in second. The difference between them is equal to 0.055. We can infer the fact that UDP is a bit more faster than TCP. The experiment number with 0, 1, 3, 4, 6 ,10, 16, 18, 20, 23, 24, 25, 28 are the outliers for TCP when we build 95 % confidence interval. For TCP, there are 13 outliers experiments. The experiment number with 2, 6, 7, 13, 20, 28 are the outliers for UDP when we build 95 % confidence interval. For UDP, there are 7 outliers. If we compare the number of outliers for TCP and UDP, we see that the data from UDP is a bit more consistent compared to data from TCP. The inital idea that comes up to our mind is that the reason for the different number of outliers may be caused by congestion/flow control of TCP which is network usage dependent . There are congestion and flow mechanisms applied in TCP which we do not apply in UDP. The congestion and flow control causes the window size to change. The change of the window size also changes the speed and causes fluctuation.

Experiment 2 - Packet Loss

For the packet loss cases where the percentage is a positive number, the best UDP constants are 1325 as payload size and 1120 as a window size.

1) 0% Packet loss

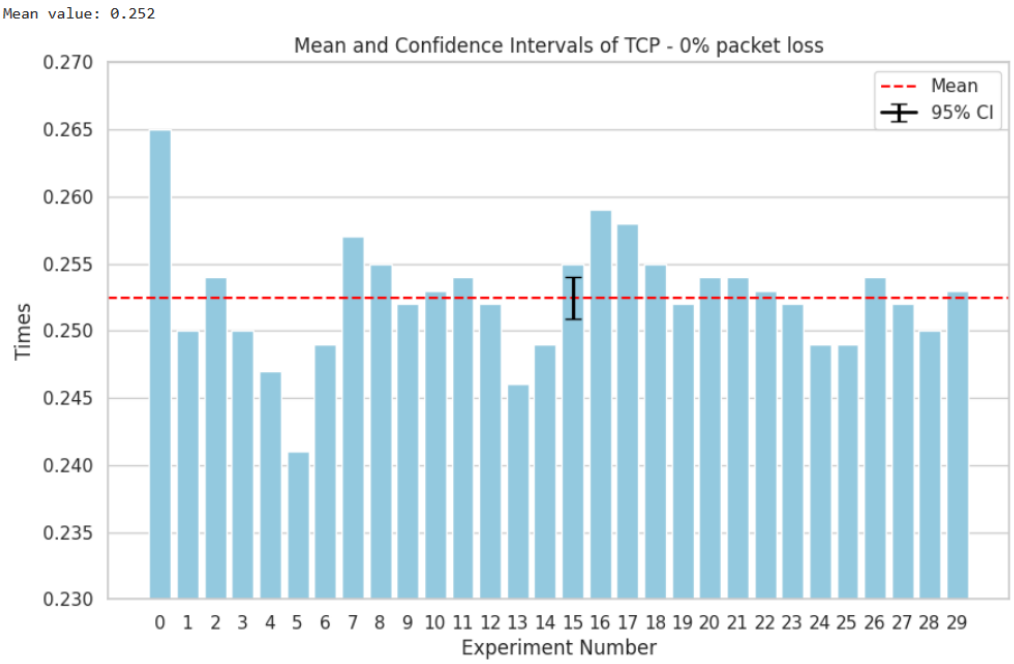

Figure 3: TCP plot (0 % packet loss tc/netem rules applied.)

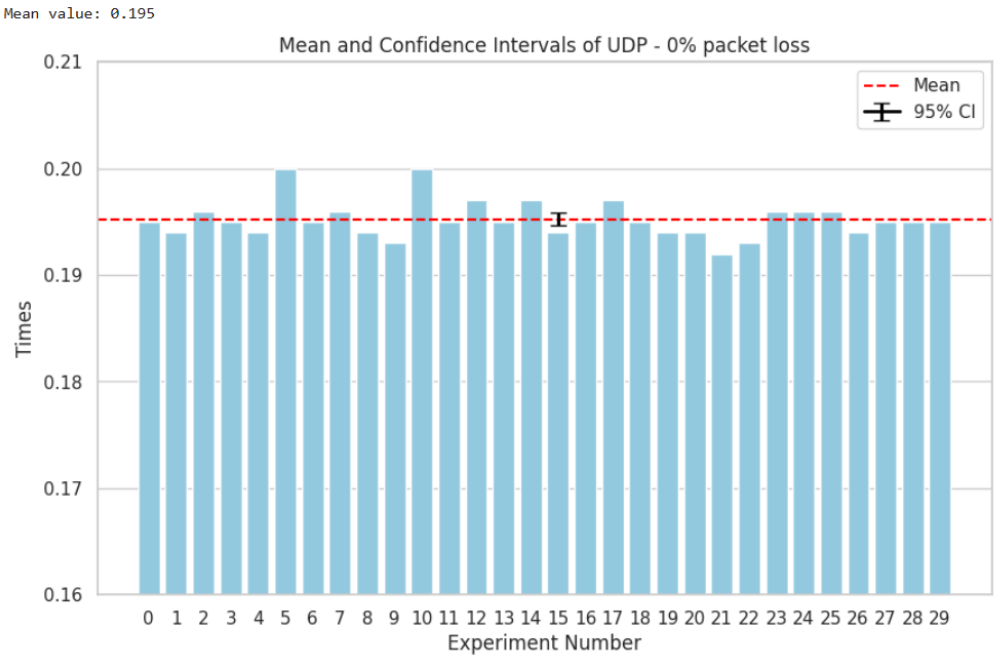

Figure 4: UDP plot (0 % packet loss tc/netem rules applied.)

2) 5% Packet loss

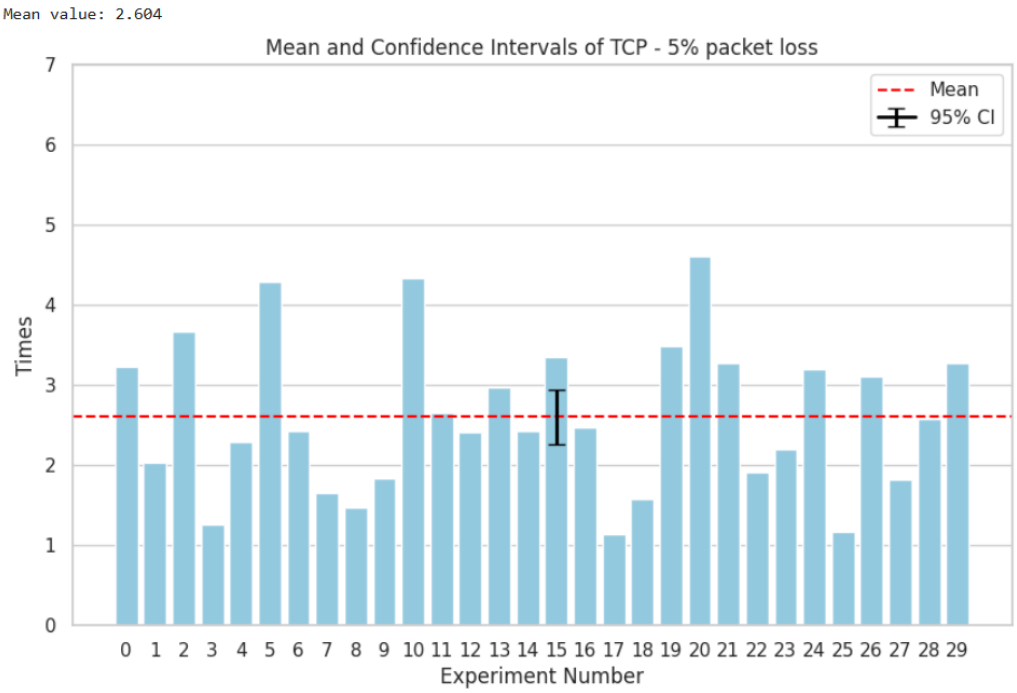

Figure 5: TCP plot (5 % packet loss tc/netem rules applied.)

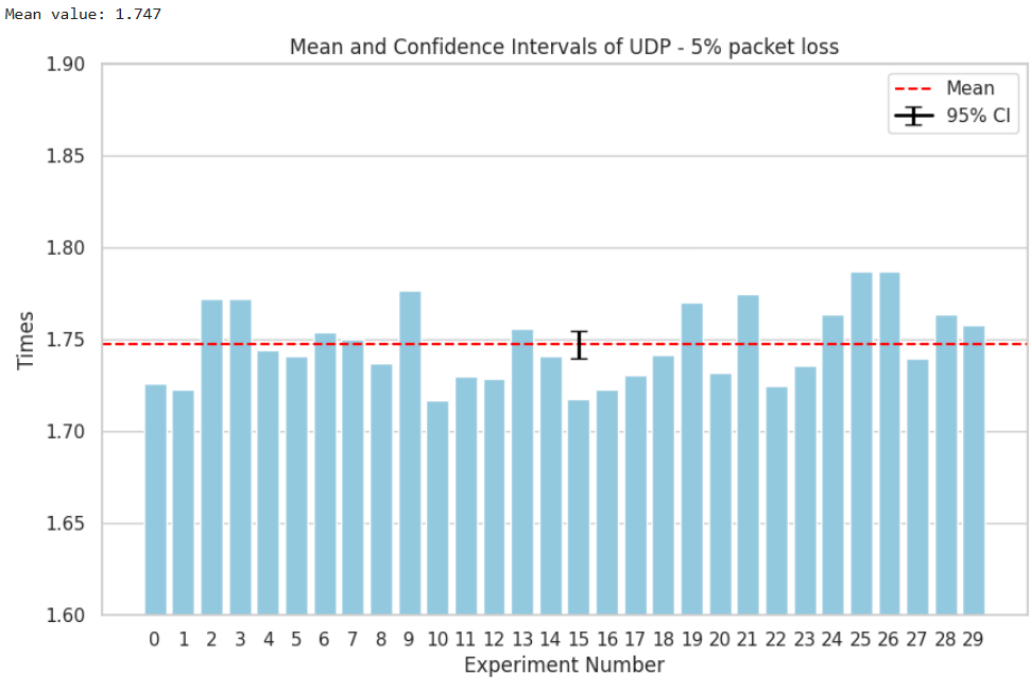

Figure 6: UDP plot (5 % packet loss tc/netem rules applied.)

3) 10% Packet loss

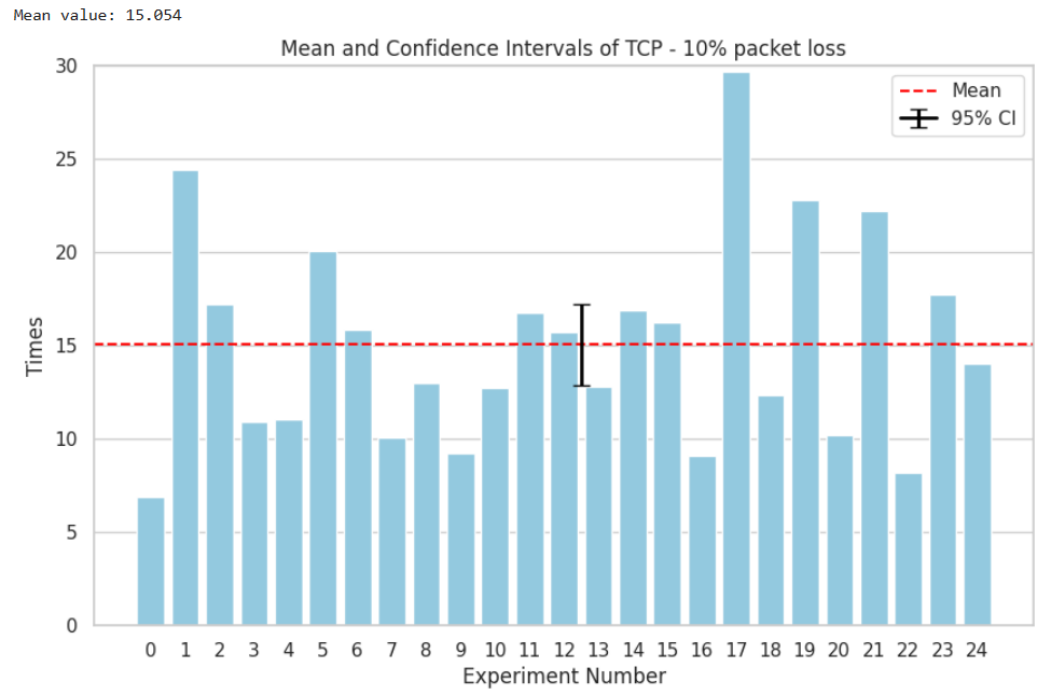

Figure 7: TCP plot (10 % packet loss tc/netem rules applied.)

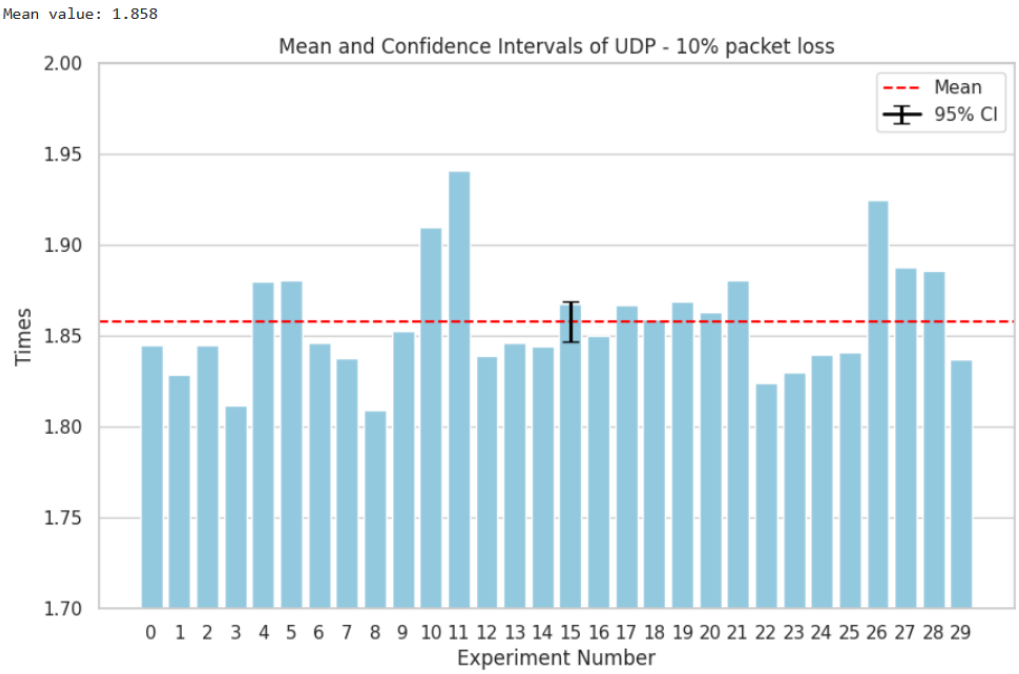

Figure 8: UDP plot (10 % packet loss tc/netem rules applied.)

4) 15% Packet loss

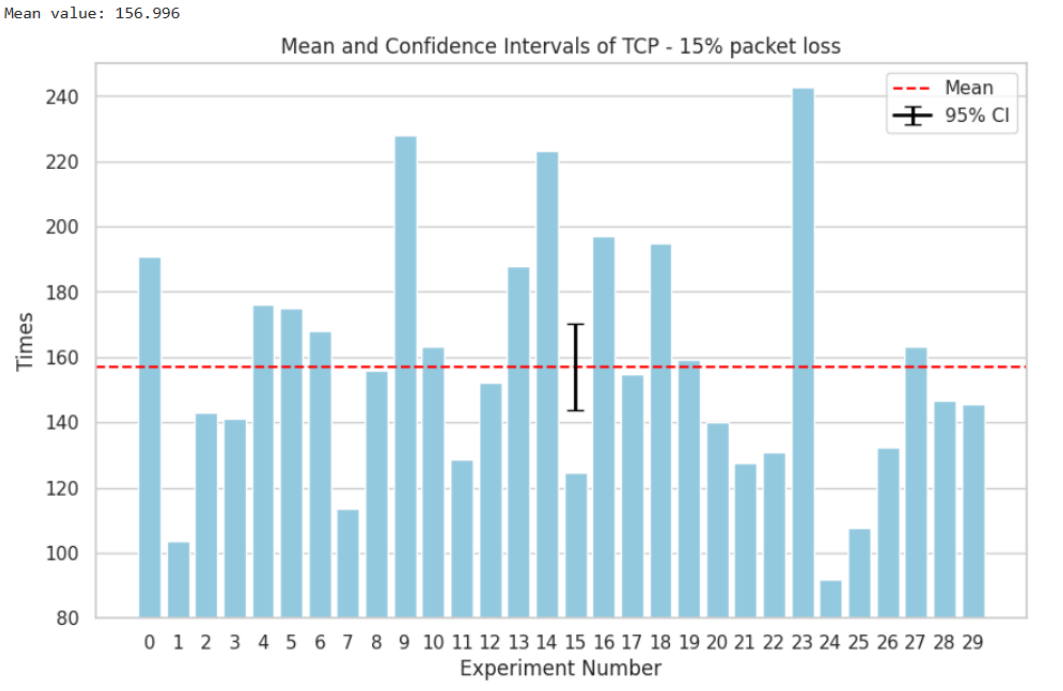

Figure 9: TCP plot (15 % packet loss tc/netem rules applied.)

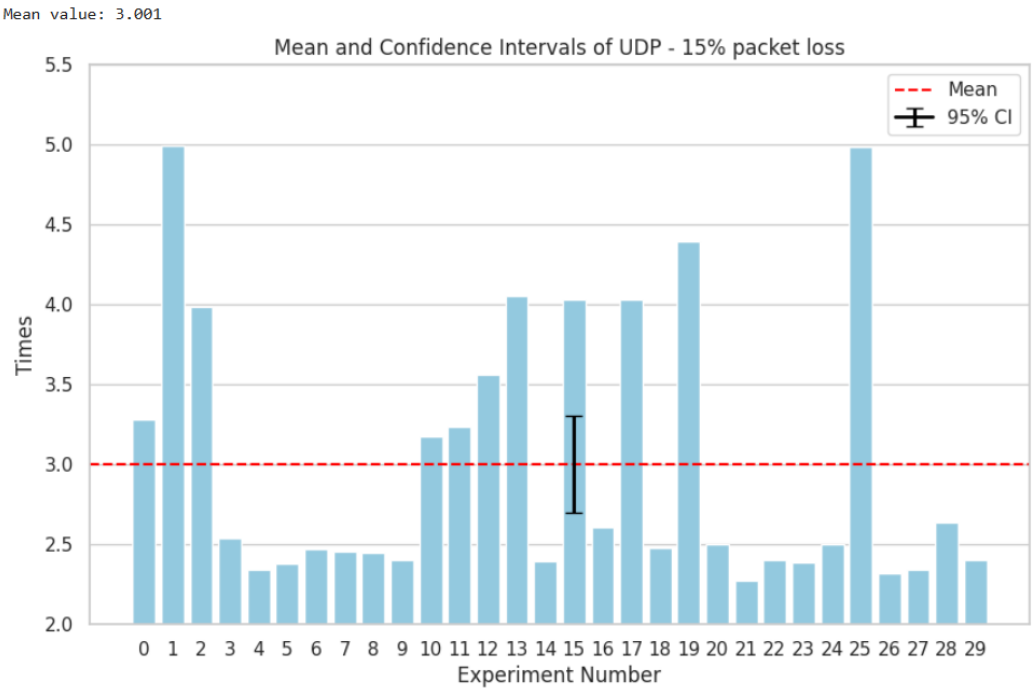

Figure 10: UDP plot (15 % packet loss tc/netem rules applied.)

Analyzing the experiment - 2 (Packet Loss)

For this experiment, UDP has used 1120 as a window size and 3455 as a payload size.

From obtained plots, we can see that packet loss affected both TCP and UDP , but also, TCP is affected much more than UDP.

If packet is lost, TCP increases the next expected timeout depending on the current packet's RTT value measured. That is why when more packets are lost, more time each timeout occurence, and hence, whole TCP takes. Additionally, when a packet is lost, TCP indicates this as a sign of congestion in the network and reduces its current window size since TCP is congestion-controlled protocol. To be detailed, the more lost packets, smaller the window size, *cwnd*, is the case for TCP.

Our UDP implementation does not have that congestion-controlled specification, therefore it is affected much less than TCP.

It is normal that both implementations are affected more with increasing packet loss percentage since it will take much more time to complete the task and fluctuates because of randomity (binomial distribution) of packet loss possibilities.

If we compare this results with the experiments below, we see that packet loss is the slowest one except for the delay experiment. This result is about transmission. If the packet is lost, that is, the packet cannot arrive the destination, we have to retransmit it according to our retranmission timeout value. This also means we have to wait for a certain amount of time before we retransmit it. We do not dynamically adjust the value of the timeout seconds according the RTT data as the TCP does, as explained above.

Another issue we observe is as the corruption increases, there are more cases where the processes either does not end or takes too much time for both TCP and UDP. These rare cases, 3 or 4 times out of 30 experiments, would probably be outlier if we included them in our statistics.

Experiment 3 - Packet Duplication

For the 0 % duplicate part, the efficient values for UDP are 21 for the window size and 10000 for the payload size. These values make UDP faster than TCP for 0 % duplicate case.

1) 0% Duplicate

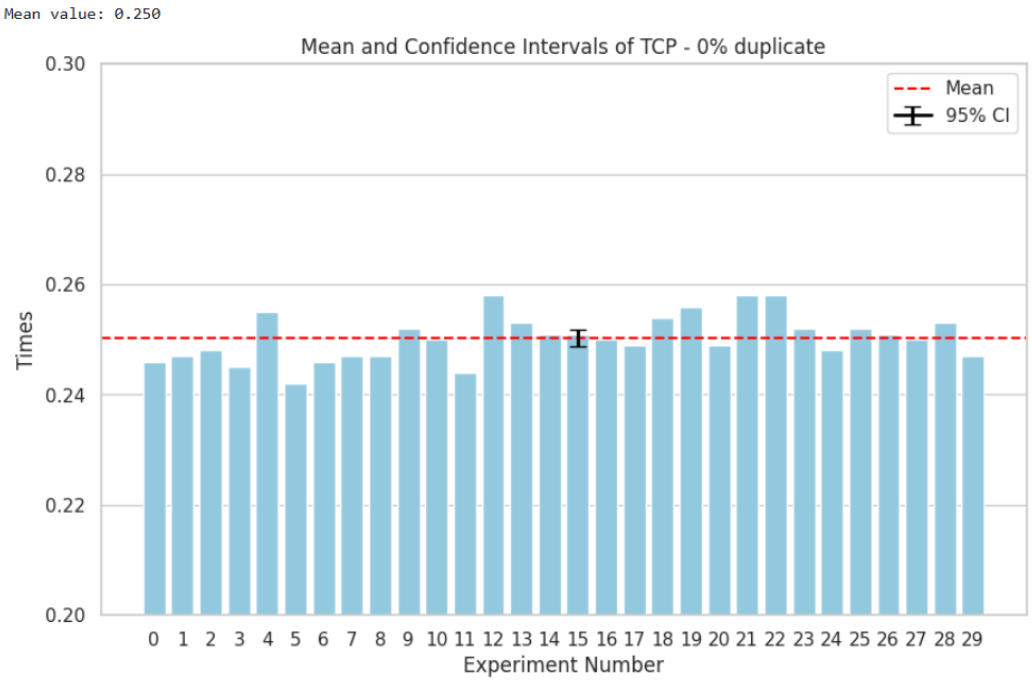

Figure 11: TCP plot (0 % packet duplication tc/netem rules applied.)

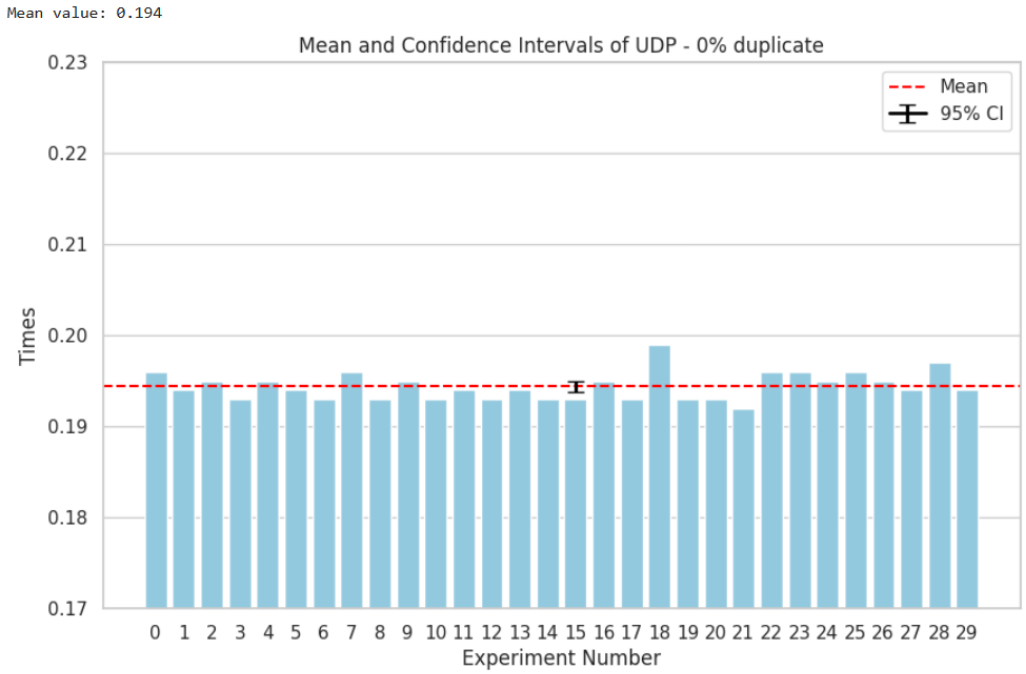

Figure 12: UDP plot (0 % packet duplication tc/netem rules applied.)

2) 5% Duplicate

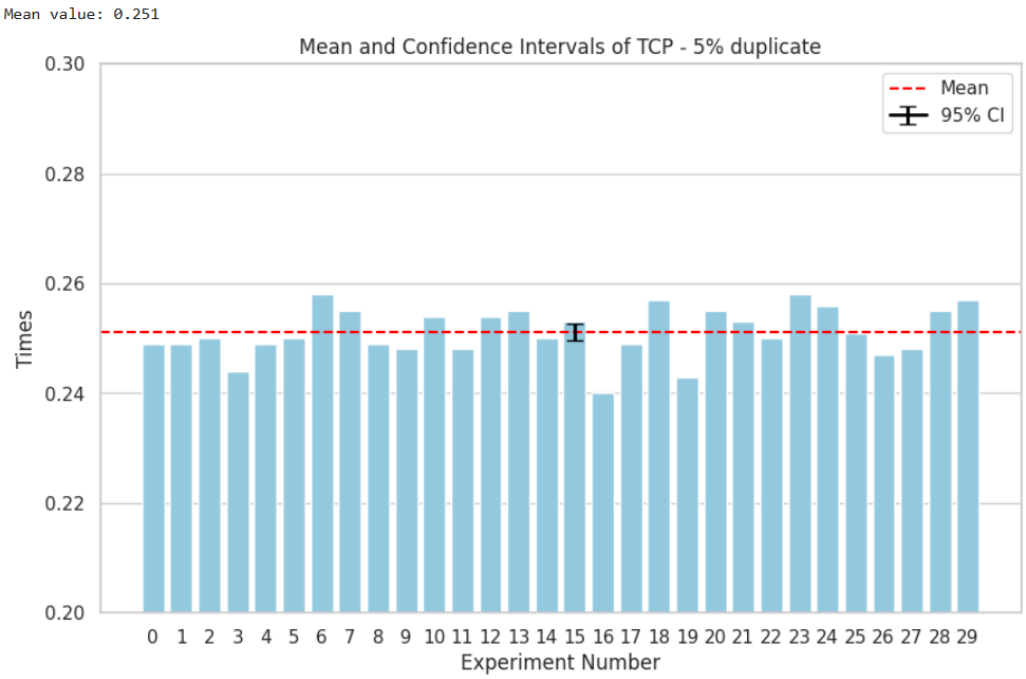

Figure 13: TCP plot (5 % packet duplication tc/netem rules applied.)

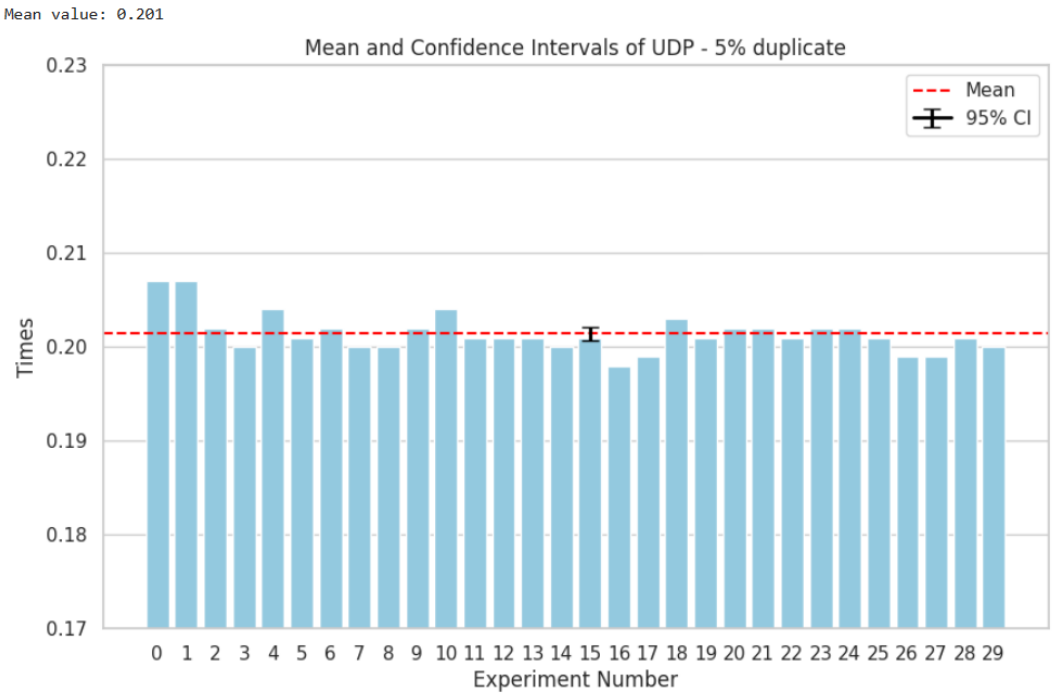

Figure 14: UDP plot (5 % packet duplication tc/netem rules applied.)

3) 10% Duplicate

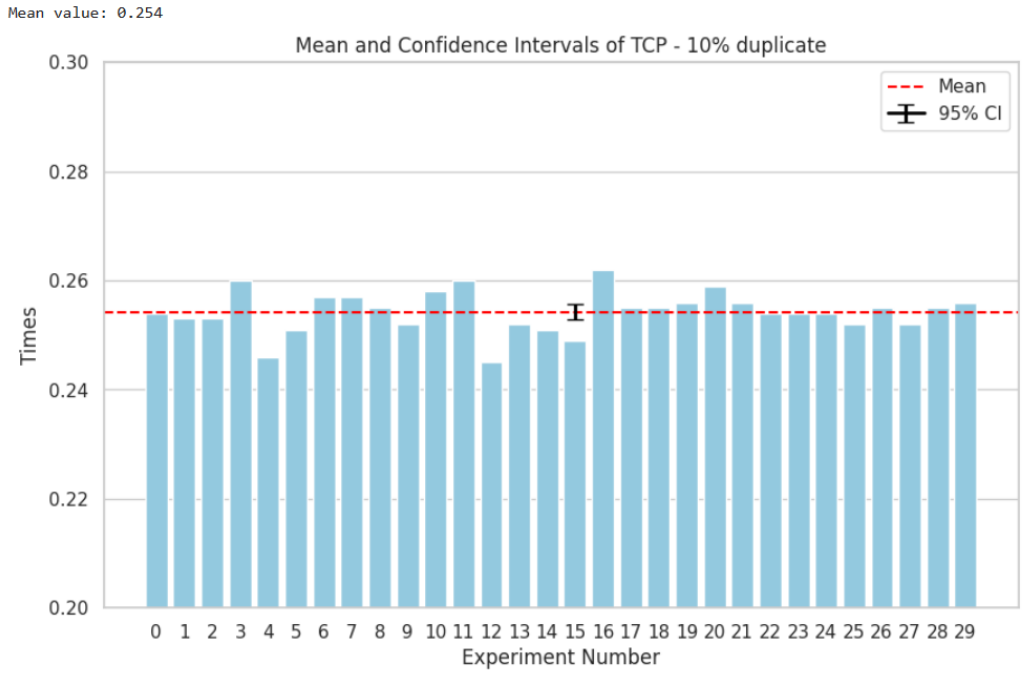

Figure 15: TCP plot (10 % packet duplication tc/netem rules applied.)

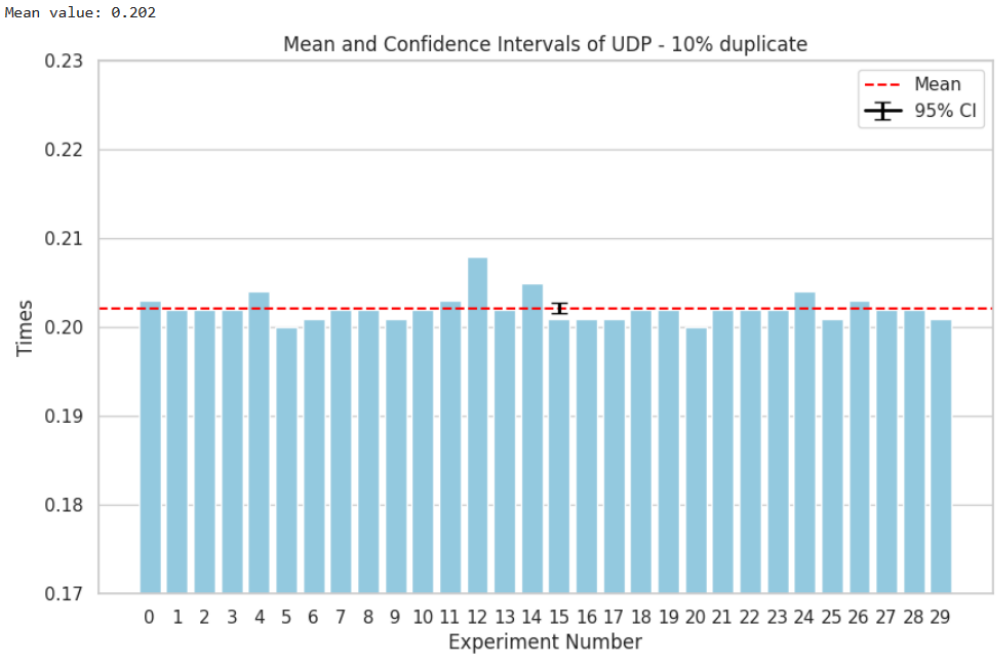

Figure 16: UDP plot (10 % packet duplication tc/netem rules applied.)

Analyzing the experiment - 3 (Duplicate Packet)

From the figures above, we can conclude that the duplicates do not have much affect since both TCP and UDP implementations just make retransmission with 3 duplicate ACKs and having a bit more packets without any additional loss does not cause much changes. The results we get are much smiliar to those we get with either 0% netem settings or the benchmark one. In the client (receiver) implementation, if the client receives a packet that is previosly acked, or a packet that is not available in the client (receiver) window, then client only ignores it. The client sends the first packet in its window as a packet. By doing that, the client says "This is the packet I am expecting for the server to send". We also consider this part as implementation depended result. We basically ignore since we want to implement selective repeat with cumulative ACK. If one does not choose to ignore, duplicate packet will probably cause their application to slow down.

Experiment 4 - Packet Corruption

1) 0 % Packet Corruption

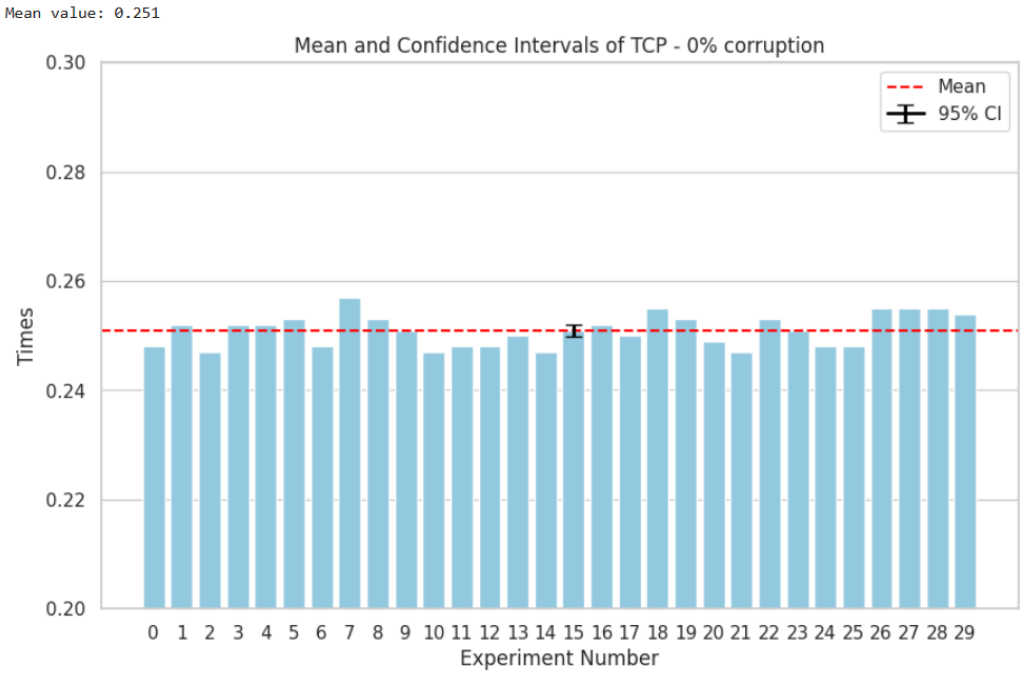

Figure 17: TCP plot (0 % packet corruption tc/netem rules applied.)

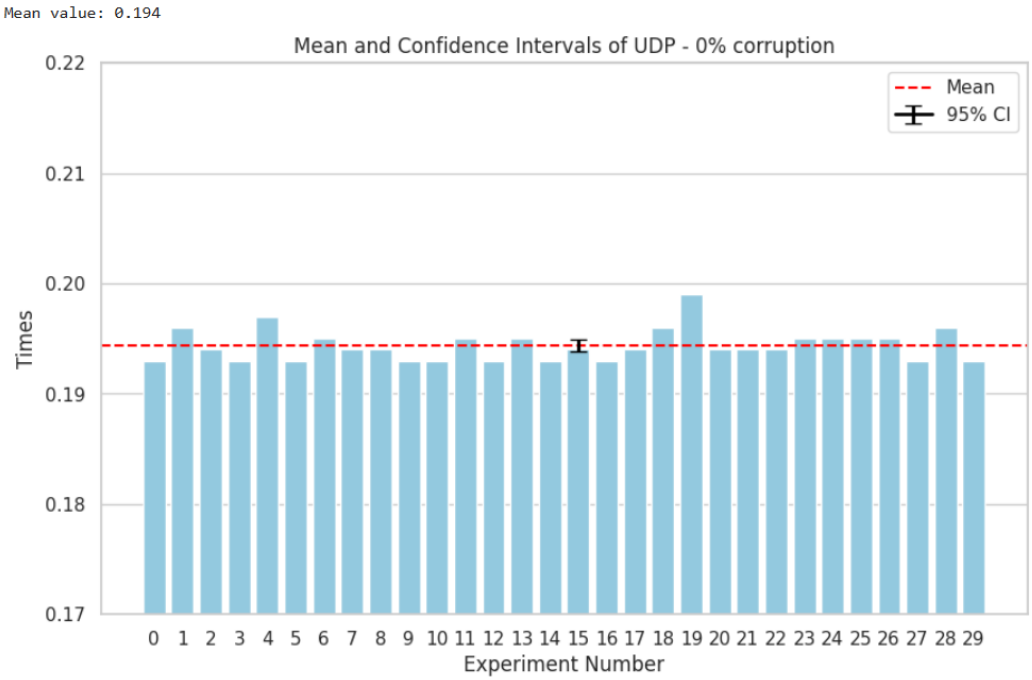

Figure 18: UDP plot (0 % packet corruption tc/netem rules applied.)

2) 5 % Packet Corruption

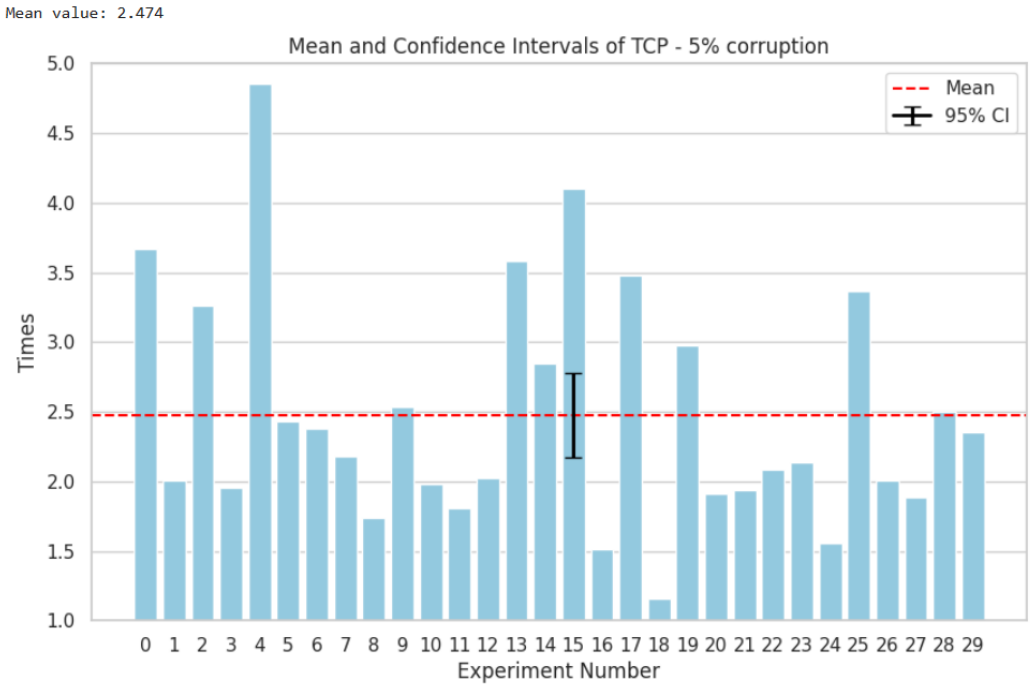

Figure 19: TCP plot (5 % packet corruption tc/netem rules applied.)

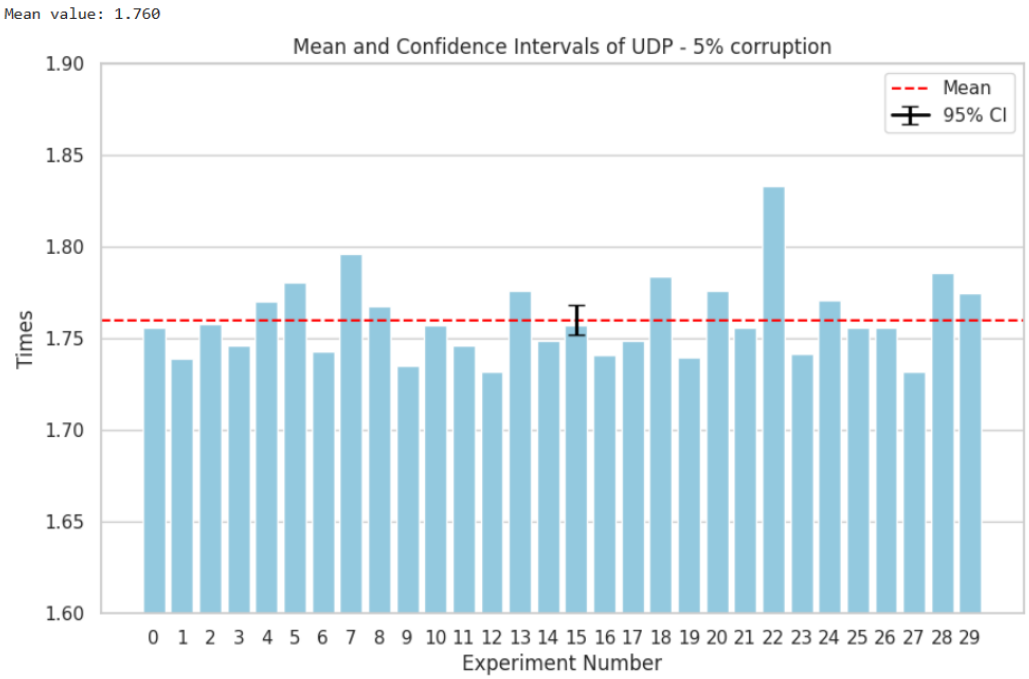

Figure 20: UDP plot (5 % packet corruption tc/netem rules applied.)

3) 10 % Packet Corruption

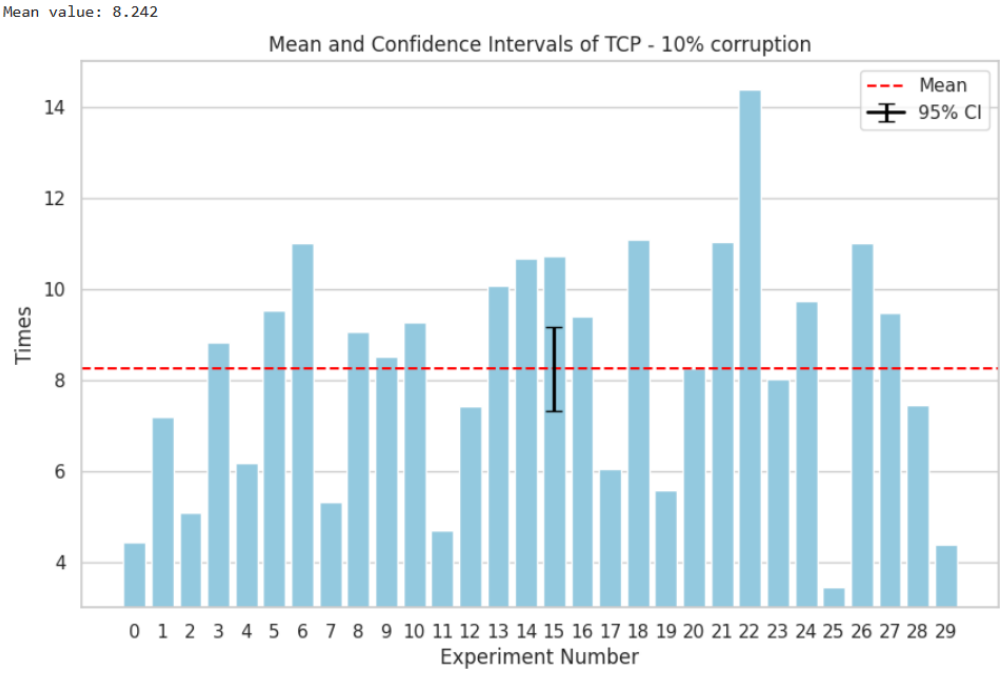

Figure 21: TCP plot (10 % packet corruption tc/netem rules applied.)

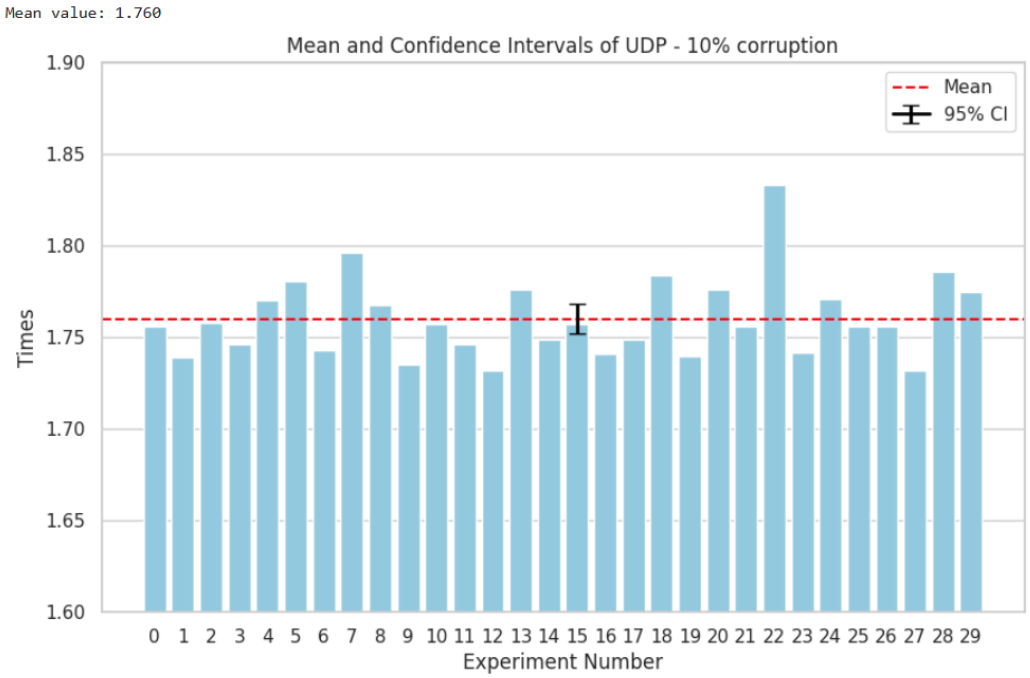

Figure 22: UDP plot (10 % packet corruption tc/netem rules applied.)

Analyzing the experiment - 4 (Packet Corruption)

The packet corruption case is similar to packet loss because we drop the packet if the pre-calculated checksums are not identical to each other. However, there is a small difference between packet loss and packet corruption case. This difference is that we can respond with ACK as a receiver faster then the packet corruption case. Even though we drop the packet, we just resend the ACK packet meaning the next sequence number expected from the receiver (client). Therefore, there will be probably less timeout cases compared to packet loss case. Timeouts are the ones that slow down the process for our project, especially for the loss case.

Experiment 5 - Delay

1) 0 % 100ms - Uniform Distribution

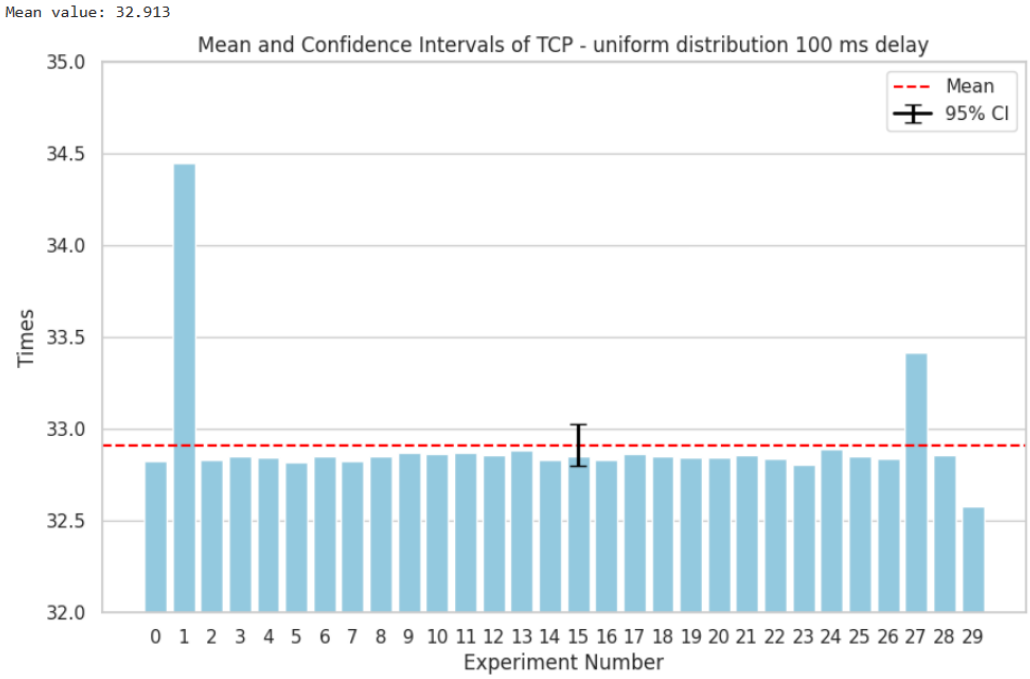

Figure 23: UDP plot (100 ms uniform tc/netem rules applied.)

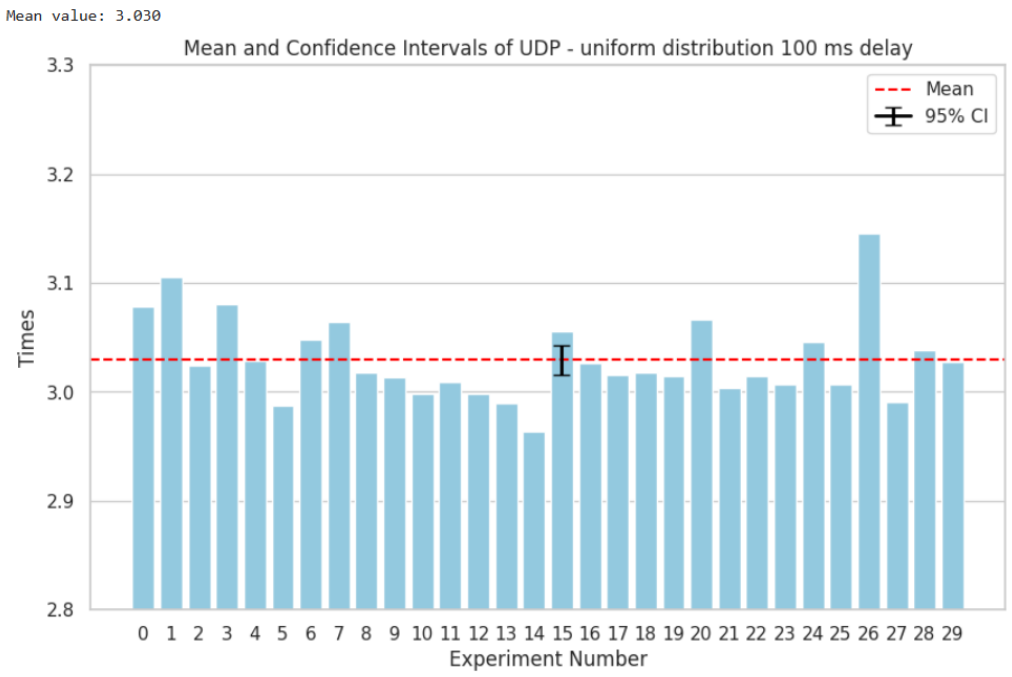

Figure 24: UDP plot (100 ms uniform - 20 ms jitter tc/netem rules applied.)

2) 0 % 100ms - Normal Distribution with jitter Value 20ms

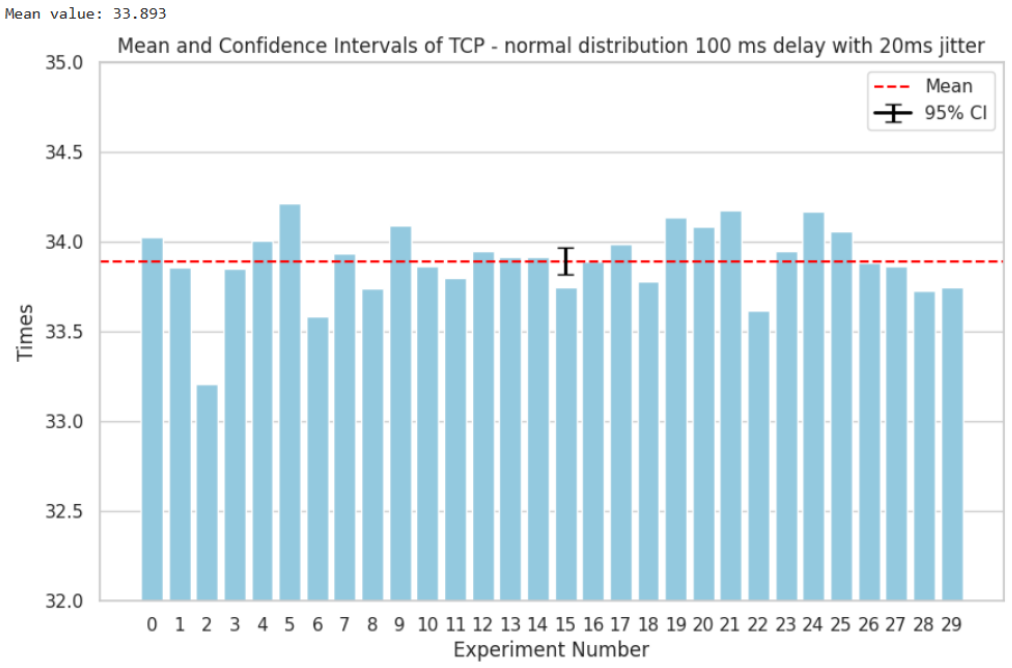

Figure 25: TCP plot (100 ms normal - 20 mis jitter tc/netem rules applied.)

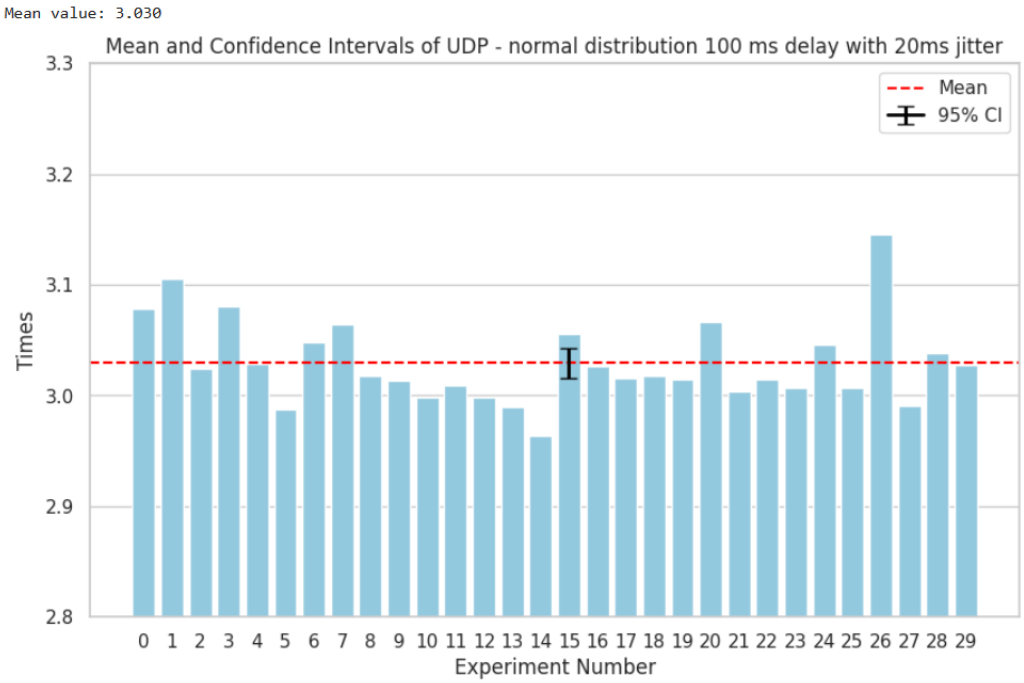

Figure 26: UDP plot (100 ms normal tc/netem rules applied.)

Analyzing the experiment - 5 Delay

For the uniform distribution, we observe less fluctuations since the uniform distribution gives us almost constant delay. However, the one with normal distribution has more fluctuations for both UDP and TCP. Here, the means for the experiments are very close to each other. Like most of the experiments above, we observe UDP is faster than TCP. TCP implements timeout duration as a variable. It updates the timeout depending on RTT (Round Trip Time). The estimated RTT is calculated with the following formula.

$$EstimatedRTT = (1 - \alpha) \times EstimatedRTT + \alpha \times SampleRTT$$

The timeout is calculated with the following formula.

$$Timeout = EstimatedRTT + 4 * DevRTT$$

These are the formulas implemented by TCP for the sake of congestion and flow control. If the timeout becomes much less than RTT, the server (sender in our case) may send unnecessary packets, which increases the congestion level of the network. Furthermore, if the timeout becomes much greater than RTT, the server (sender in our case) may be prone to wait for unnecessary amount of time, which slow downs the process. These means we need timeout that is close to RTT within some deviation. However, we do not implement these in our Reliable Data Transfer over UDP. Our implementation sends the packets regardless of the current situation of the either network or the receiver, or sender. We are not interested in whether the containers are not ready for receiving the packet or whether the network is congested and so on.