CENG 435

Data Communications and Networking

Fall 2023-2024

Programming Assignment

Full Name: **Meriç KARADAYI**
Student ID: **2448553**
Full Name: **Ahmet Eren KARDAŞ**
Student ID: **2448579**

1 Introduction

In this assignment, we will implement two types of socket programming protocols: Unreliable Data Protocol (UDP) in a Reliable Data Transfer (RDT) fashion, and Transfer Control Protocol (TCP). Our goal is to send ten large (1MB) and ten small (10KB) objects between the server and client under various network conditions, such as possible loss, corruption, delay, etc.
In our UDP implementation, we will design UDP to function like RDT, employing techniques similar to Selective Repeat. On the other hand, in our TCP implementation, we will not introduce any additional features. It operates as TCP normally does since its principles inherently ensure reliable data transfer. The server functions as the sender, while the client acts as the receiver.
We will have test our implementations 30 times per experiment in different network conditions, and will compare our results with respect to conditions and each other.

2 Implementation details

2.1 UDP

2.1.1 Server

Basic algorithm is as following. After getting an "Hello message" from a client. Initially, server is reading all files one-by-one with one large and one small order and append this read bytes to a list which is going to contain all files. After each file is read completely, server also adds an EOF to indicate the boundaries of the objects. Then first server creates packets for all elements of the list which contains all files. This packet consist of 3 fields, checksum for checking whether the sent objects is correct, sequence number for keeping track of objects, and the data which is part of some file. Since the all files in the list are added sequentially in the all file list we do not need to keep the file name information. After all packets are generated, server firstly sends the total packet number to the client to inform client that how many packets client is going to expect. After a response specific to this packet is received. It is ready to send packets. If such response does not come, server continously sends packet number to client with some time intervals. Later on, a dictionary is used for storing to responses of sent packets with keys are the sequence

numbers and the values are boolen that indicates whether that packet with sequence number is ACKed. The rest of the algorithm is similar to Selective Repeat with a main loop.
Sending Packets: Server checks dictionary for non-ACKed packets and sends N packets. N value is the window size which we have decided after some runs. If the non-ACKed packets are less than N, it can send same packet multiple times. This helps to algorithm at the end of files. When server is traversing in dictionary, it also keeps the last traversed sequence number. In next iteration it starts looking from the last traversed sequence. This provides shifting of windows in each iteration. When a packet is not acked server sends that packet again after sending all other packets. This helps when there is a delay of arrival of packets.
Receiving ACK responses: After N packets are sent. Server is expecting to take N packets as response. If a timeout occurs during receiving this responses, it continues with the next iteration by again sending packets. If packets are recieved in wanted time, server controls the sequence number of the coming packet and it marks that packet as received in ACKed packets dictionary. When an response is received to an already ACK packets, server ignores it. If received packet contains a sequence number specific to all files are acked or all sequence numbers in dictionary is marked, it ends the all process.

2.1.2 Client

UDP starts with sending a Hello message to the server to provide its own address. The client continuously sends this hello message until any response is received from the server. Client side has a main loop running. It waits for a packet. Then tries to parse the coming packet. If any error occurs during this parsing or receiving it discards that packet. After packet is received in success, it calculates the checksum from incoming packet data and check with checksum in the packet. If they do not match, again it discards. It firstly waits for a packet with sequence number specified to total packet number and creates a dictionary with respect to that number with similar to server side. After that dictionary is created. Client checks the sequence number of received packet. If that packet is recieved first time, that packet data is added to dictionary with that sequence number and sends an ACK response to server. If the packet already exists in the dictionary, client sends an ACK message again to inform the server that it is added in client side. When all packets are added to dictionary, client quits from process and sends a packet with sequence number spesific to all files are received.
After all files are received, client splits the received datas from EOF's and creates seperate files for all objects.

2.2 TCP

2.2.1 Server

Server waits for a connection with a client. When connection is established, server reads and send all files with 1024 bytes chunks. Like in our UDP implemetation it adds and EOF to end of all files.

2.2.2 Client

Client first connects with server. Then receives packets from the socket until all data are received. Client adds all data to a mega object and than splits that mega object from EOFs and writes to seperate files.

3 Experiments

3.1 Benchmark

In this section, we will evaluate our two protocols in an environment without any specific rules. These experiments will serve as a benchmark for subsequent evaluations.

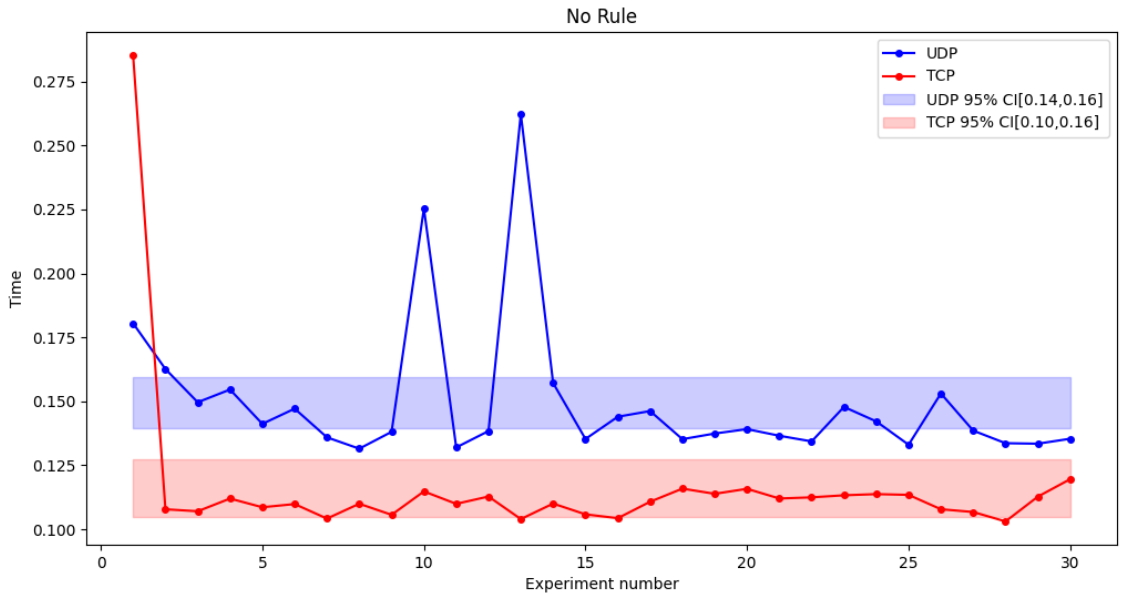

Figure 1: Benchmark

After conducting our tests, the values obtained are presented in Figure-1. Based on Figure-1, we can conclude that in our implementations without any specific environmental conditions, our protocols yield similar results. There are instances where we observe peaks in the figure, which can be stem from consecutive tests.

3.1.1 TCP

Regarding our TCP implementation, we can assert that the results are satisfactory, showing consistent distributions. However, since these results will serve as a benchmark, it is not appropriate to categorize them as good or bad in terms of speed.

3.1.2 UDP

Concerning our UDP implementation, we can note that we achieve a commendable speed compared to TCP in tests conducted without any specific environmental conditions. This is noteworthy because UDP is inherently unreliable, and mitigating data loss, corruption, or delays often requires sacrificing some efficiency. As a result, when UDP is implemented as RDT with careful considerations, it seems to exhibit slower performance in no condition in graph, but it can be seen that the time interval in time axis is relatively small.

3.2 Packet Loss

In this experiment, we will assess our two protocols in an environment where conditions lead to packet loss during file transfer. We conducted tests 30 times for each condition, with environment conditions set at 0%, 5%, 10%, and 15% packet loss.

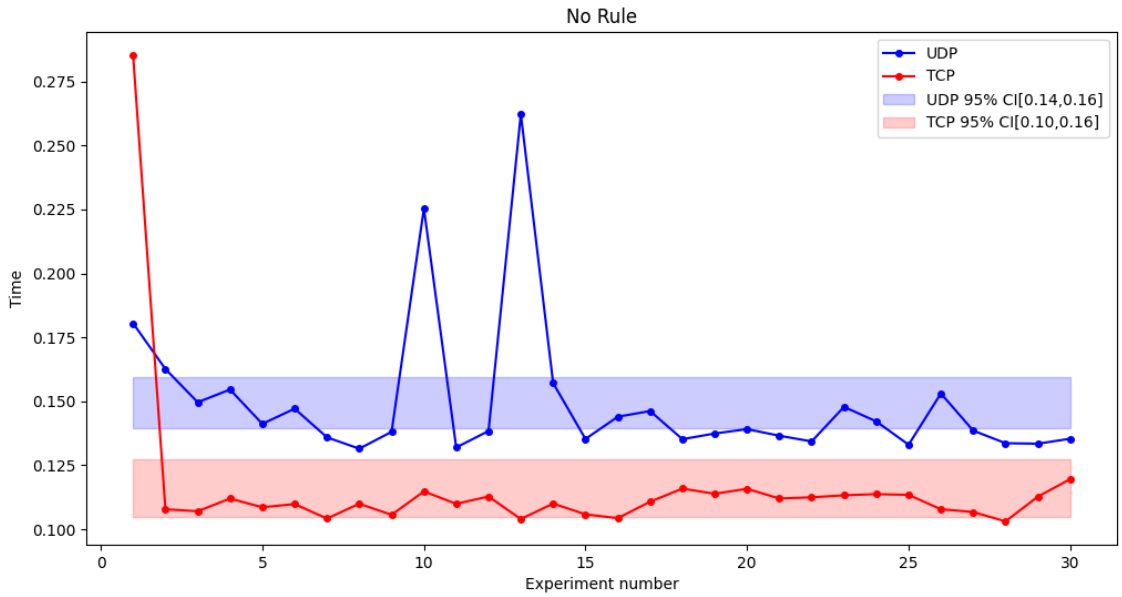

Figure 2: Packet Loss: %0 / Benchmark

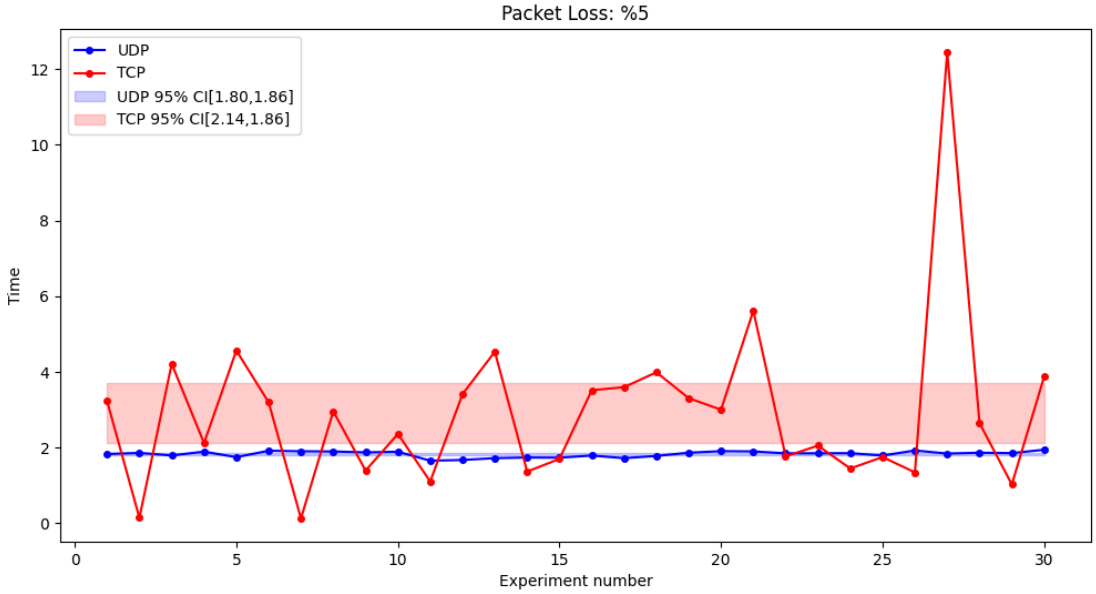

Figure 3: Packet Loss: %5

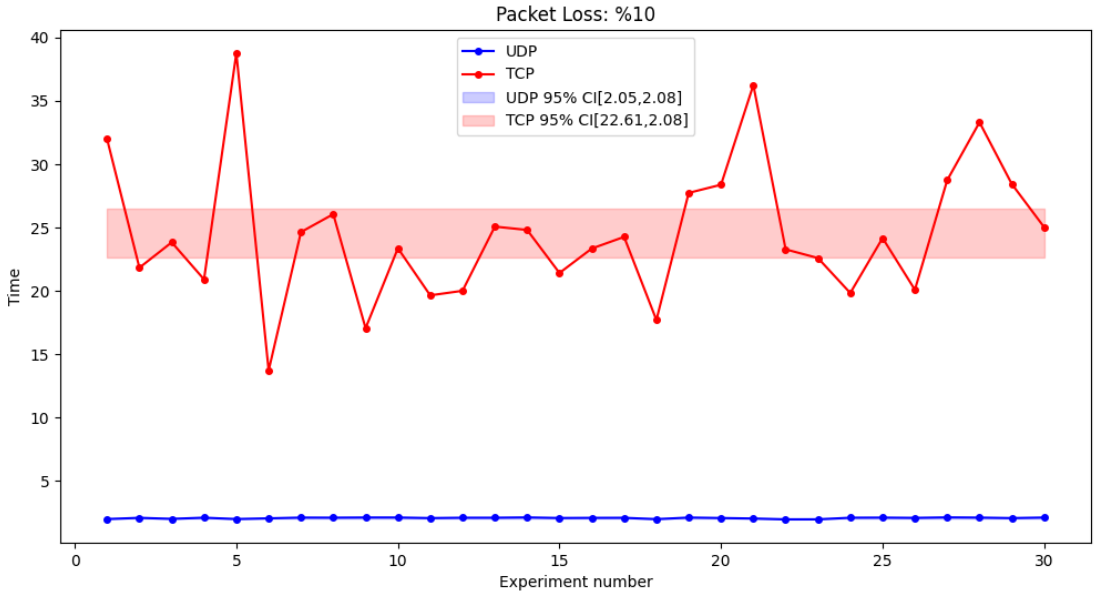

Figure 4: Packet Loss: %10

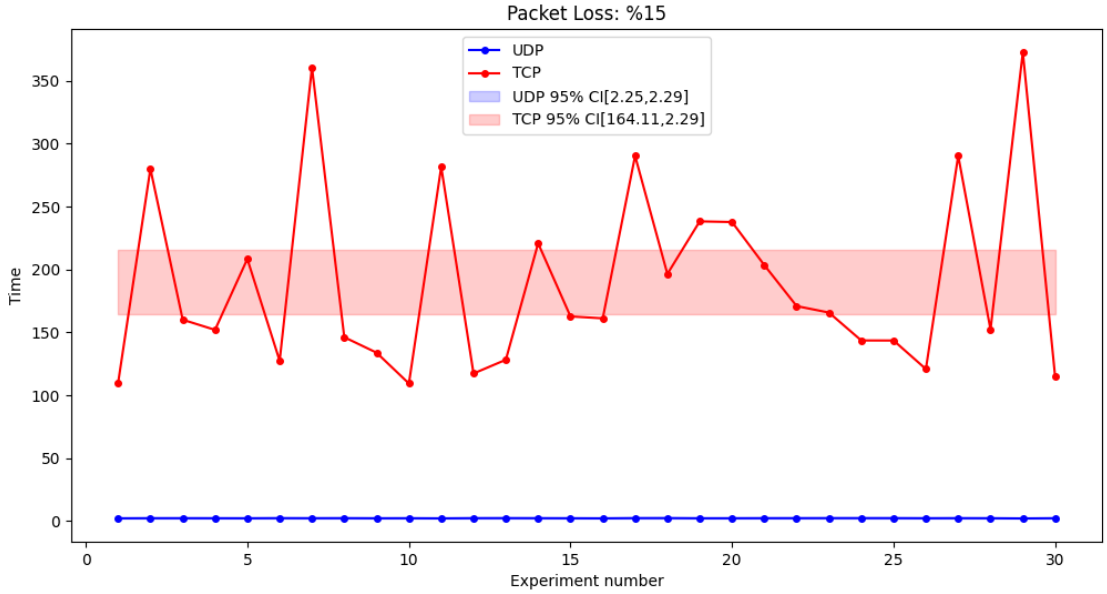

Figure 5: Packet Loss: %15

After conducting our tests, the obtained values are presented in Figure 2 as the benchmark, Figure 3 for 5% packet loss, Figure 4 for 10% packet loss, and Figure 5 for 15% packet loss. From the experiment results depicted in the figures above, we can draw the following conclusions:
As expected, when comparing these experiments with the benchmark results, the total time values increased significantly with every increment in packet loss percentage. Handling packet loss proves to be

	%0 Loss	%5 Loss	%10 Loss	%15 Loss
UDP	0.149	1.83	2.067	2.268
TCP	0.116	2.928	24.54	190

Table 1: Mean Values

time-consuming in both UDP and TCP.

.

3.2.1 TCP-UDP Comparison

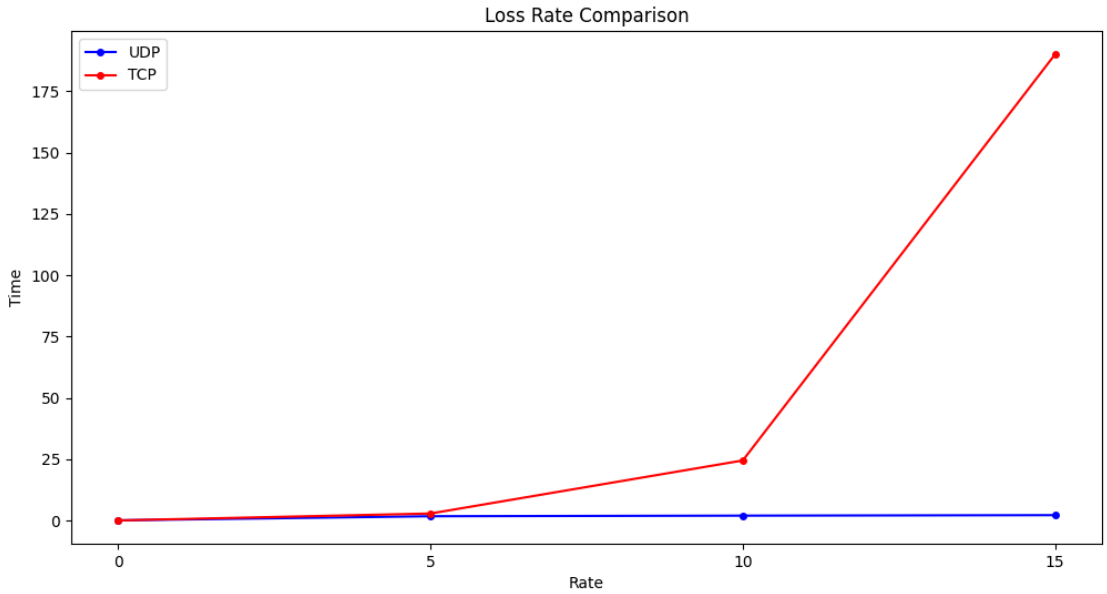

Figure 6: Loss Rate Comparison

When we compare UDP and TCP in a packet loss environment, our UDP implementation demonstrates an advantage. In UDP, there is no need to wait or spend time handling packet loss; instead, it continues to send and receive packets continuously by shifting the window. In contrast, TCP is a connection-oriented and reliable protocol, equipped with mechanisms for error detection, retransmission of lost packets, and ordering of received packets. When TCP detects packet loss, it initiates a retransmission of the lost packet, introducing delay in the communication process. In our implementation, we did not provide any system to enhance TCP's speed. Consequently, in an environment with significant packet loss, our UDP implementation works significantly faster than TCP since it does not try to get packet correctly and then continue.

3.3 Packet Duplication

In this experiment, we will assess our two protocols in an environment where conditions lead to packet duplication during file transfer. We conducted tests 30 times for each condition, with environment conditions set at 0

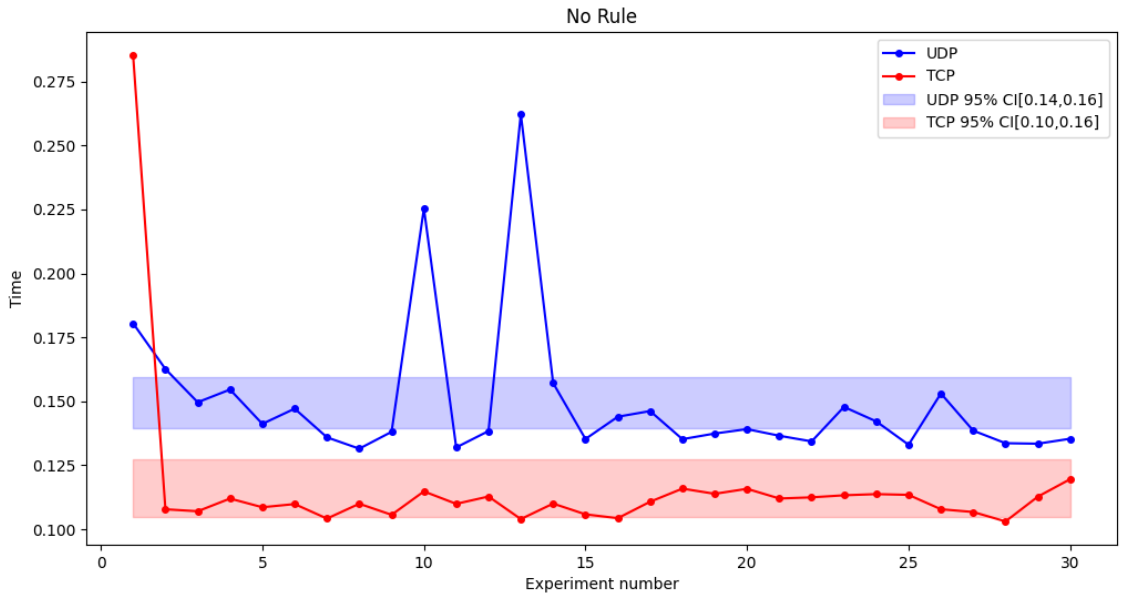

Figure 7: Packet Duplication: %0 / Benchmark

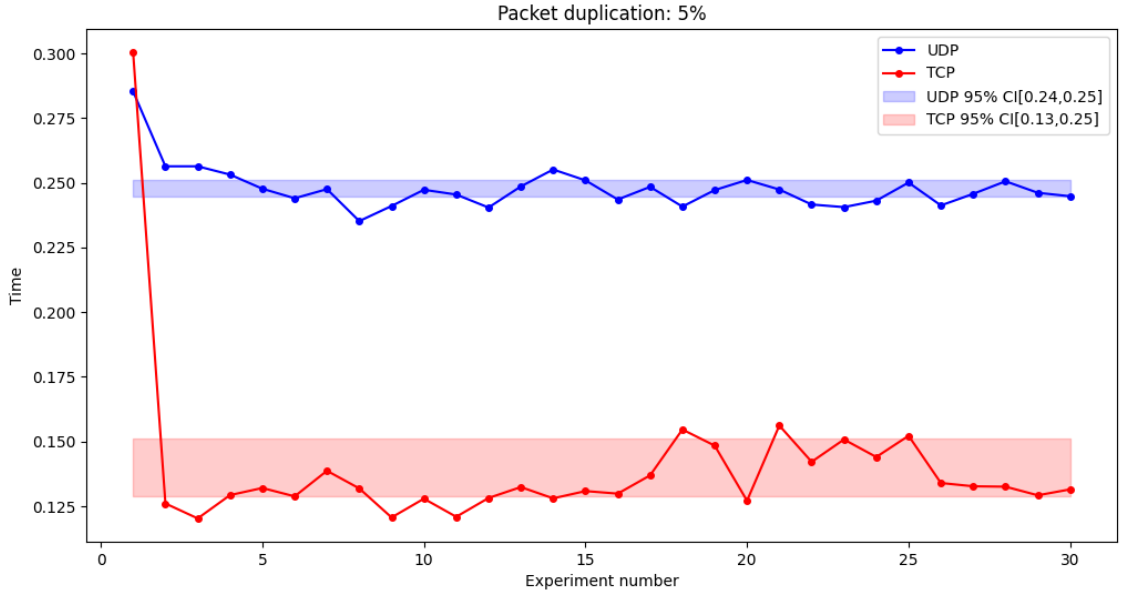

Figure 8: Packet Duplication: %5

	%0 Dup	%5 Dup	%10 Dup
UDP	0.149	0.247	0.268
TCP	0.116	0.139	0.136

Table 2: Mean Values

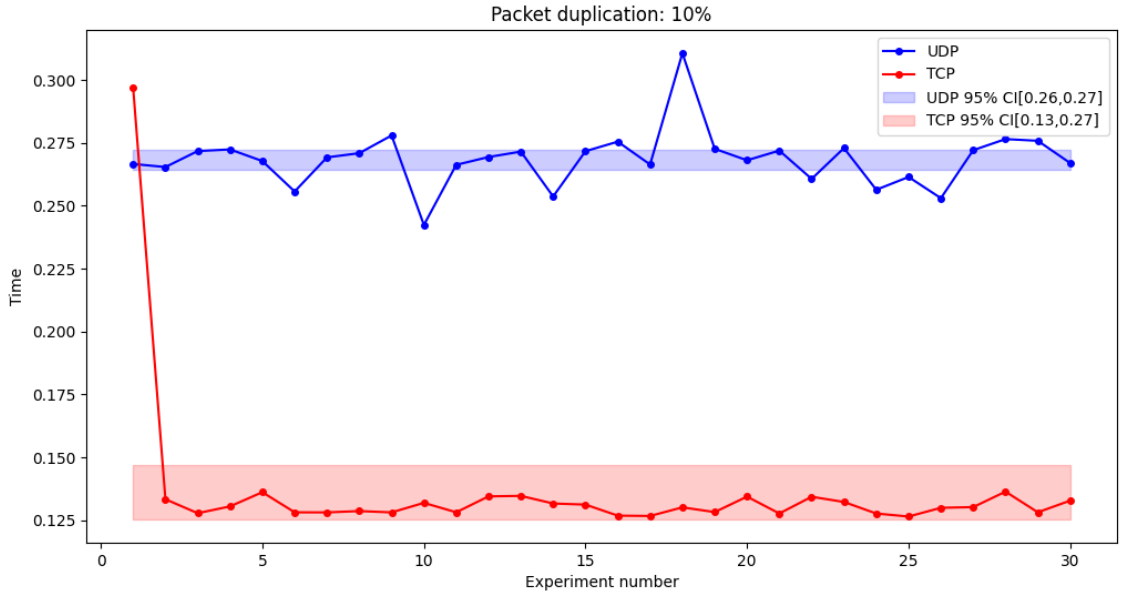

Figure 9: Packet Duplication: %10

After conducting our tests, the obtained values are presented in Figure 7 as the benchmark, Figure 8 for 5% packet duplication and Figure 9 for 10% packet duplication. From the experiment results depicted in the figures above, we can draw the following conclusions:

Comparing these results with the benchmark results, we observe that while the total time values increased in an environment where duplications were allowed, TCP results were not significantly affected by duplications.

3.3.1 TCP-UDP Comparison

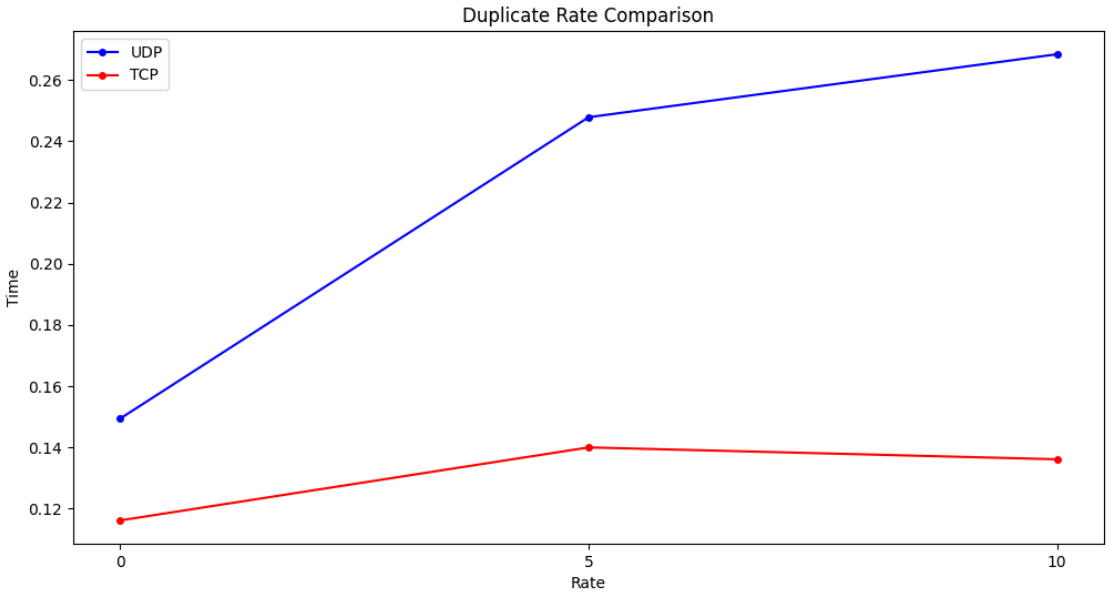

Figure 10: Duplicate Rate Comparison

We can conclude that the inherent TCP duplication handling mechanism works effectively in these experiments and environmental conditions compared to our RDT implementation in UDP. The reason for this could be that our duplication handling mechanism operates at the application level, whereas TCP's duplication handling occurs in lower layers. Consequently, the handle cost of TCP's mechanism is lower than our UDP mechanism.
Moreover, it is noteworthy that the increase in duplication percentage in the environment did not linearly cause a proportional increase in total times. When comparing Figure 7 and Figure 8, it becomes apparent that duplications do impact the time spent managing packets. However, due to our implementation, the increase in duplication did not significantly affect the overall time in our experiments.

3.4 Packet Corruption

In this experiment, we will assess our two protocols in an environment where conditions lead to packet corruption during file transfer. We conducted tests 30 times for each condition, with environment conditions set at 0%, 5%, 10% packet corruption.

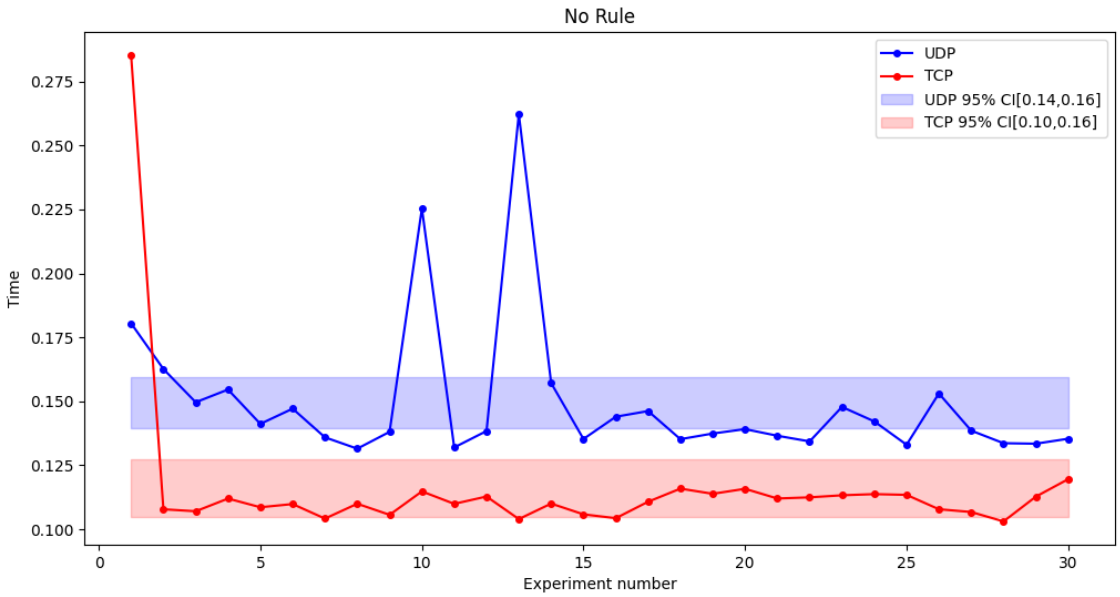

Figure 11: Packet Corruption: %0 / Benchmark

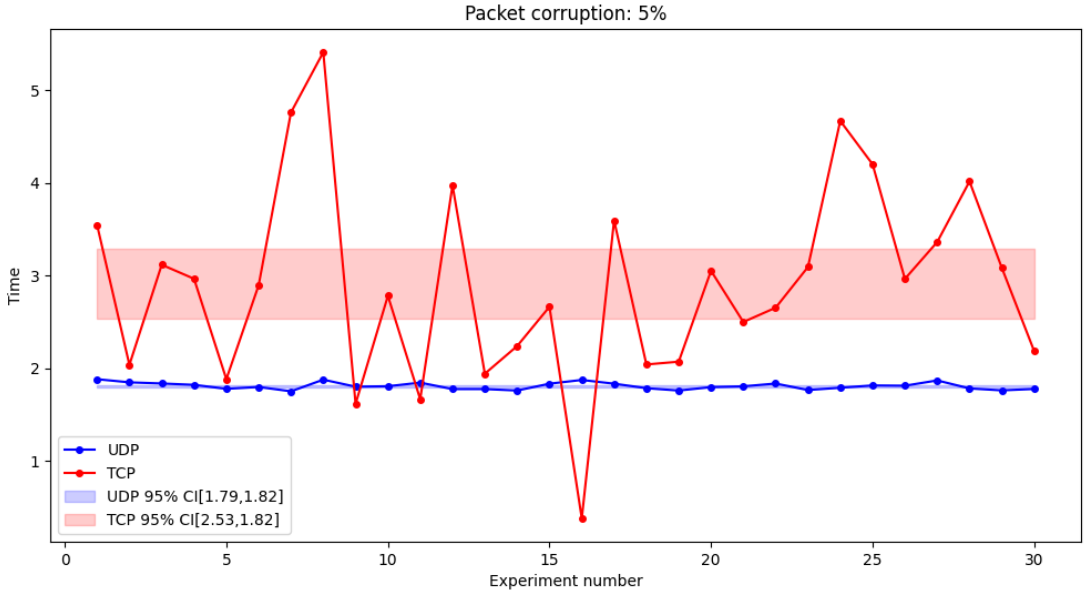

Figure 12: Packet Corruption: %5

	%0 Corr.	%5 Corr.	%10 Corr.
UDP	0.149	1.8	2.07
TCP	0.116	2.91	27.9

Table 3: Mean Values

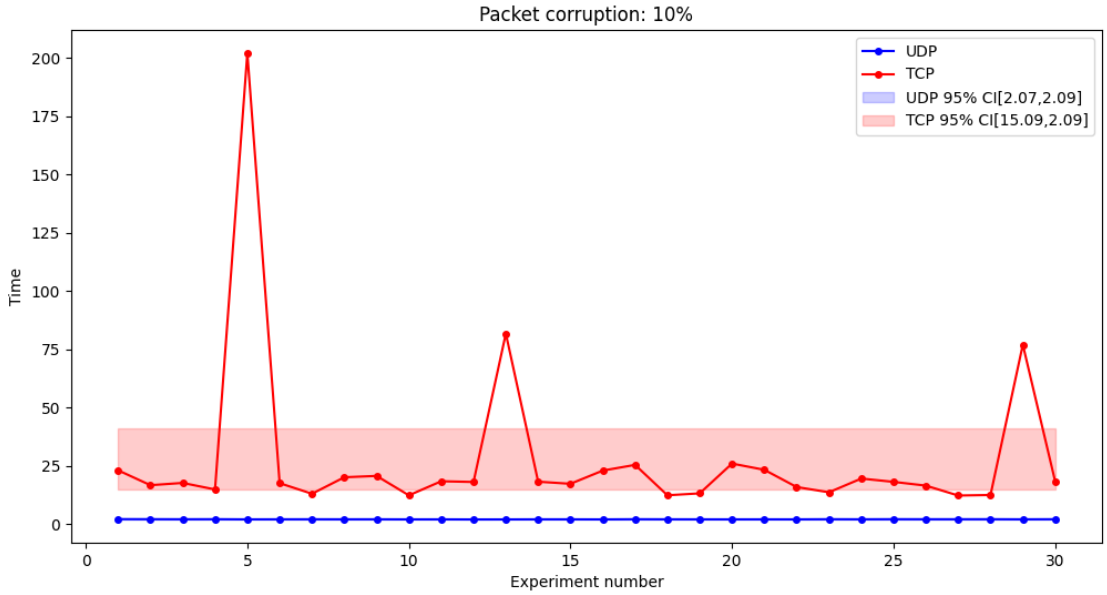

Figure 13: Packet Corruption: %10

After conducting our tests, the obtained values are presented in Figure 11 as the benchmark, Figure 12 for 5% packet corruption and Figure 13 for 10% packet corruption. From the experiment results depicted in the figures above, we can draw the following conclusions:
As expected, when comparing these experiments with the benchmark results, the total time values increased with every increment in packet corruption percentage. Handling packet corruption proves to be time-consuming in both UDP and TCP.

3.4.1 TCP-UDP Comparison

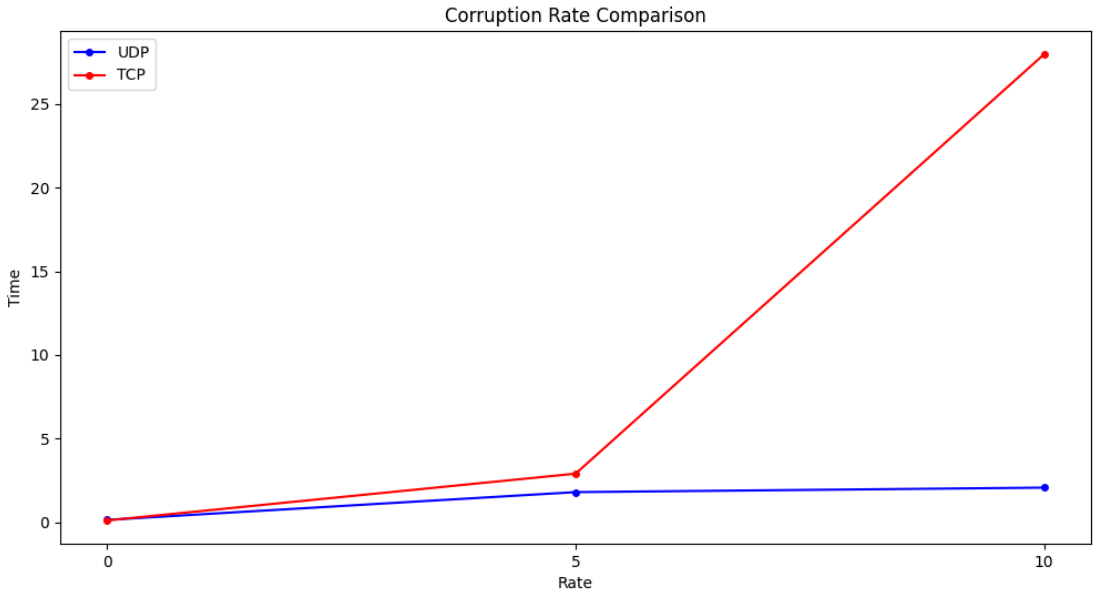

Figure 14: Loss Rate Comparison

When comparing UDP and TCP in a packet corruption environment, our UDP implementation demonstrates an advantage once again. TCP comes with its own error-handling mechanisms, and being a reliable protocol, it handles these errors in some way. In contrast, our UDP implementation, while handling packet corruption, doesn't attempt to immediately correct the corrupted packet. Instead, it continues to send packets from other files, allowing us to avoid wasting time waiting for the correction of corrupted packets. In summary, our UDP implementation doesn't wait to correct corrupted packets and keeps sending other packets, resulting in significant performance compared to TCP's own corruption handling mechanism.

3.5 Packet Delay

In this experiment, we will assess our two protocols in an environment where conditions lead to packet delay during file transfer. We conducted tests 30 times for each condition, with environment conditions set at 100ms - uniform distribution, 100ms - normal distribution delays.

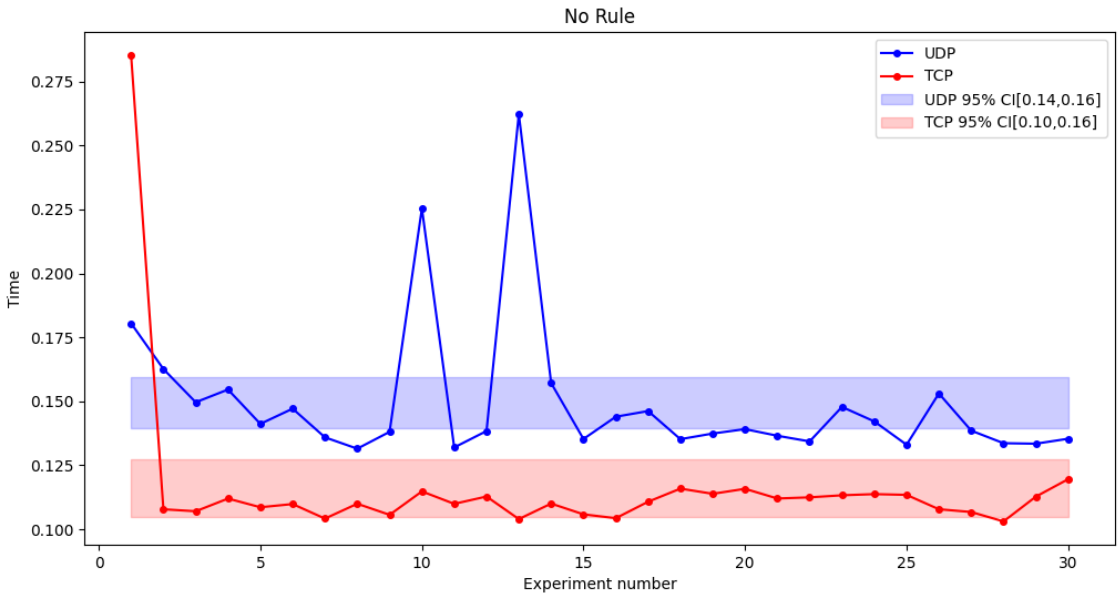

Figure 15: Benchmark

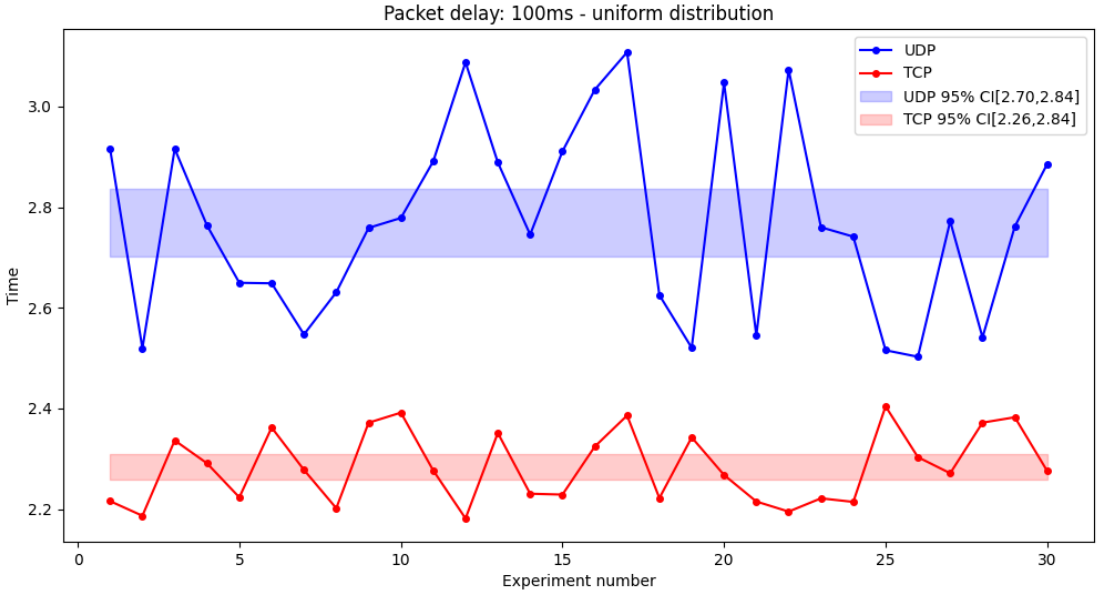

Figure 16: 100ms - uniform distribution

	Uniform	Normal
UDP	2.769	3.07
TCP	2.284	7.065

Table 4: Mean Values

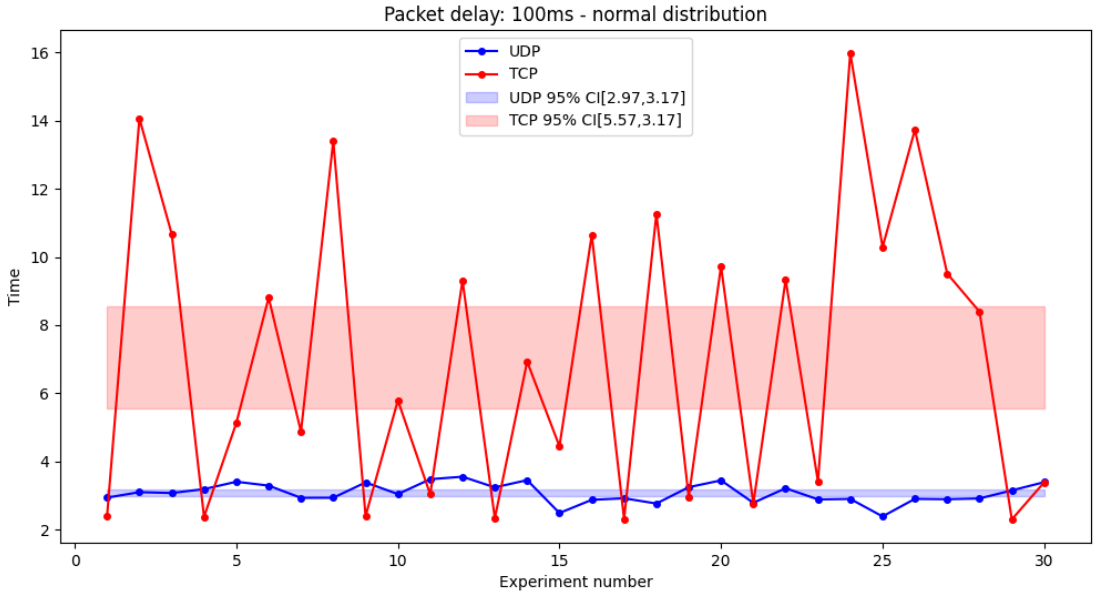

Figure 17: 100ms - normal distribution

After conducting our tests, the obtained values are presented in Figure 15 as the benchmark, Figure 16 for 100ms - uniform distribution and Figure 17 for 100ms - normal distribution. From the experiment results depicted in the figures above, we can draw the following conclusions:

As expected, in any type of delay, total file transfer times increase. As we could anticipate, with higher delays in a curved normal distribution, both protocols are much slower in a normal distributed delay environment compared to a uniform one. Finally, we can also say that while our UDP implementation's performance does not change significantly between distributions, TCP, on its own, is much slower in a normal distribution.

3.5.1 TCP-UDP Comparison

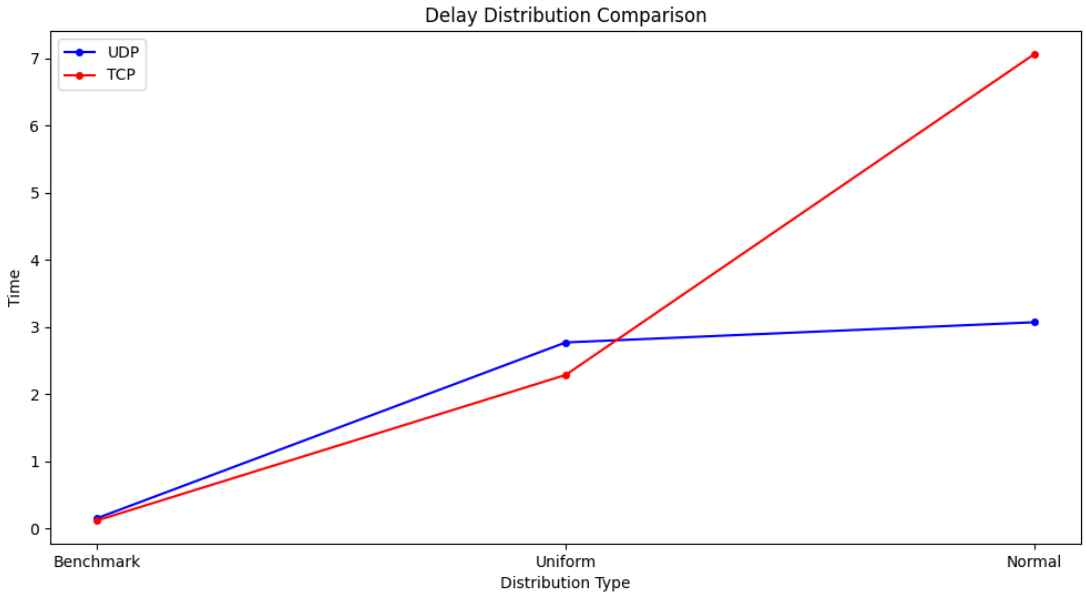

Figure 18: Delay Distribution Comparison

When comparing the two implementations, TCP's performance varies significantly between distributions. This difference could result from our implementation, particularly the way our sending and receiving chunk sizes, and these higher chunk sizes may cause possible higher delays' effects on the completion of files. On the other hand, since we send all files in an interleaved fashion, the delay's effect on our total time is more consistent and better than TCP in situations with higher delays. Moreover, these performances can be optimized by using specific values for chunk size and timeout length to specific delay types.

4 Conclusion

With using all the experiments we have done above, we can merge all of their results and derive final conclusion between conditions and protocols. Therefore, we can see how the error handling mechanism behave in different environments conditions and how can we set our protocol more optimized in specific environments.

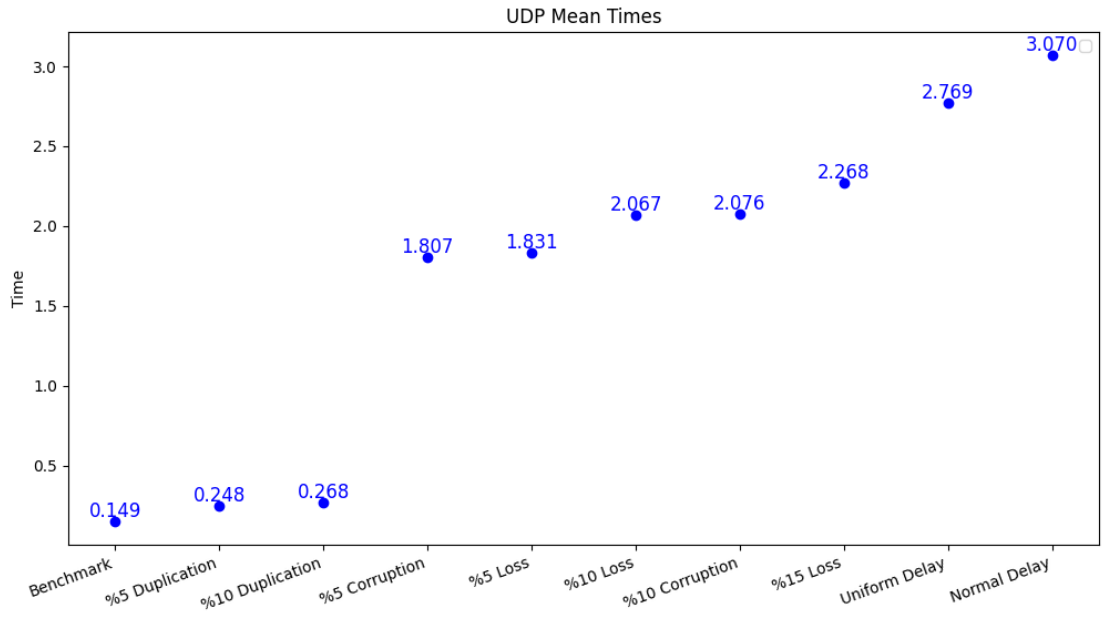

Figure 19: UDP Mean Times Comparison

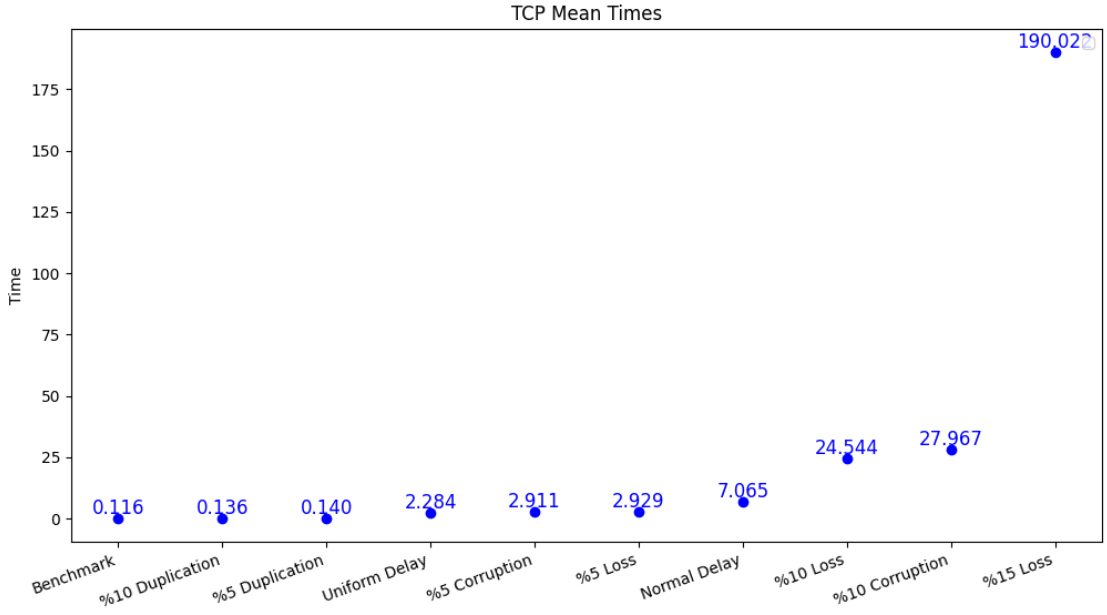

Figure 20: TCP Mean Times Comparison

From the figures above, it's evident that various types of environmental conditions have distinct effects on the protocols we've implemented. Furthermore, these figures help us comprehend the weaknesses of our algorithms in different conditions.

Beginning our conclusion by generalizing the effects of conditions, we can observe that any type of error-possible condition results in an increase in the total file transfer time. Therefore, it's safe to say that dealing with errors in any form consumes a considerable amount of time.

Furthermore, it is evident that the impact of errors such as packet corruption and packet loss results in more time consumption during handling. Additionally, delays have a considerable effect on the overall time, as expected.

Among the errors considered in the above experiments, Packet Duplication is observed to be less time-consuming. It's worth noting that fixing duplicated packets is relatively more challenging than simply discarding unnecessary duplicated packets.

Also, we observe a significant increase in the case of packet loss. Handling packet loss may be notably challenging in certain conditions. One possible reason for this is that TCP can handle corrupted packages by checking the checksum internally. However, when a packet is lost, there must be some delay to realize that the packet has not been received by the receiver. This delay can lead to a significant increase in the total time.

Finally, aside from the experiment results, we can enhance the optimization of our algorithms by adjusting variables such as window size, chunk size, timeout length, etc. Modifying these values allows us to achieve improvements in the performance of the algorithms.

4.0.1 TCP-UDP Comparison

When we compare our implementations, it's evident that our error handling algorithm used in UDP exhibits superior performance over TCP's own error handling mechanisms. The interleaved chunk sending system we employed proves to be more effective in error handling time. Instead of attempting to correct and resend packages, the approach of continuing to send packages yields better performance.