Middle East Technical University

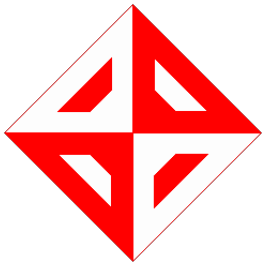
Data Communications and Networking

# Programming Assignment

*Author:*
Mert Sağcan *(2310449)*

*Author:*
Taha Emir Gökçegöz *(2448421)*

January 5, 2024



# 1 Environment

After considering the hardships of setting the project up on windows and mac devices, we decided to install and setup a virtual machine (virtual box). After installing that we followed the instructions on the Github repository and set up our environment.(docker) To make sure our environment works, we tested it by running the example codes.

# 2 TCP Implementation

## 2.1 Research

First we started with examining the example code provided to us in the template. After that we searched the book for deeper knowledge about TCP implementations. After that we also searched the internet for diagrams and for future improvements.

## 2.2 Version 1

the first version of TCP we implemented a simple sending mechanism that sends all files by segmenting the data to 512 bytes and send it all in a for loop. This version worked fine on benchmark but the timing was really slow (40 seconds).

## 2.3 Version 2

After researching more, we found the sendall() function. This function takes a single json, and sends it. Ths worked a lot better than the previous one but we thought there was still room for improvement.(0.25-0.3)

## 2.4 Version 3

After some confusion, we thought that we had to handle multiple clients at once, for that we implemented threading. For every client we created a thread that sent all the data. This version worked as fast as the second version of the code.

## 2.5 Final Version

Finally we went back at our first design and improved its speed (0.15-0.2) we changed the packet size and the handling algorithm. After hat we tested tcp on all test versions (loss, delay, duplicate and corrupt) and recorded the timings.

Listing 1: How we switched to segmenting.

```
# Send serialized data
chunk_size = 10240  # Define an appropriate chunk size
bytes_sent = 0
while bytes_sent < len(serialized_data):
    to_send = serialized_data[bytes_sent:bytes_sent + chunk_size]
    sent = conn.send(to_send)
    if sent == 0:
        raise RuntimeError("Socket connection broken")
    bytes_sent += sent
# conn.sendall(serialized_data)
```

### 2.5.1 Reading files with TCP

In the final version we implemented a case specific code that reads the ".obj" files from the directory and sends all data at once. For this we used os library.

# 3 UDP

## 3.1 Research

We spent most our time in this section. After finishing TCP, we started brainstorming on how to write a reliable pipelined UDP that overcomes head-of-line blocking. After some discussion we decided to implement RDT3.0 over UDP for starters.

## 3.2 Version 1

Our first version just implemented RDT3.0 over UDP with go-back-n logic. This made our UDP implementation way slower than our TCP implementation.

## 3.3 Version 2

In this version, we implemented threading for listening ACKs. It seemed to work well so we decided to stick with this. Also for this we tried to implement selective repeat algorithm to overcome head-of-line blocking. To sum up, our final UDP implementation takes all files and sends the data at once. We have a thread that listens to ACKs and a main thread sends the segments one by one.

# 4 Comparing TCP and UDP

In this section, we are going to elaborate on how our TCP and UDP implementations compare in different test scenarios. Main purpose of this homework is to create a reliable, pipelined protocol for UDP sockets while maintaining its speed advantage over TCP. Hence, we implemented a secure and fast protocol for UDP and tested it together with our TCP implementation. There are various test cases such as 5% and 10% package corruption, 100ms uniform and normally distributed delay etc. We made use of custom netem rules to create our testing environments.
The main reason that we do these tests is that we need robust and reliable protocols. That is why we test our implementations over various real world situations.

## 4.1 Benchmark Testing

This test were made in a sterile environment. There were no traffic, no delay and almost no loss. Purpose of this test was to check whether our UDP implementation is faster than our TCP implementation over a traffic free network.

After we tested our implementations, we have observed that our UDP implementation have become a little bit slower than our TCP implementation with only 50 ms difference overall. The main cause for this is that we have implemented so many logic on a normal UDP that in a traffic free environment, TCP have become faster. In the benchmark test, there were no loss, no corruption or duplicate and no delay. Because of that there were no need for RDT or selective repeat. We have come to this decision after testing many different versions of UDP and TCP implementations.

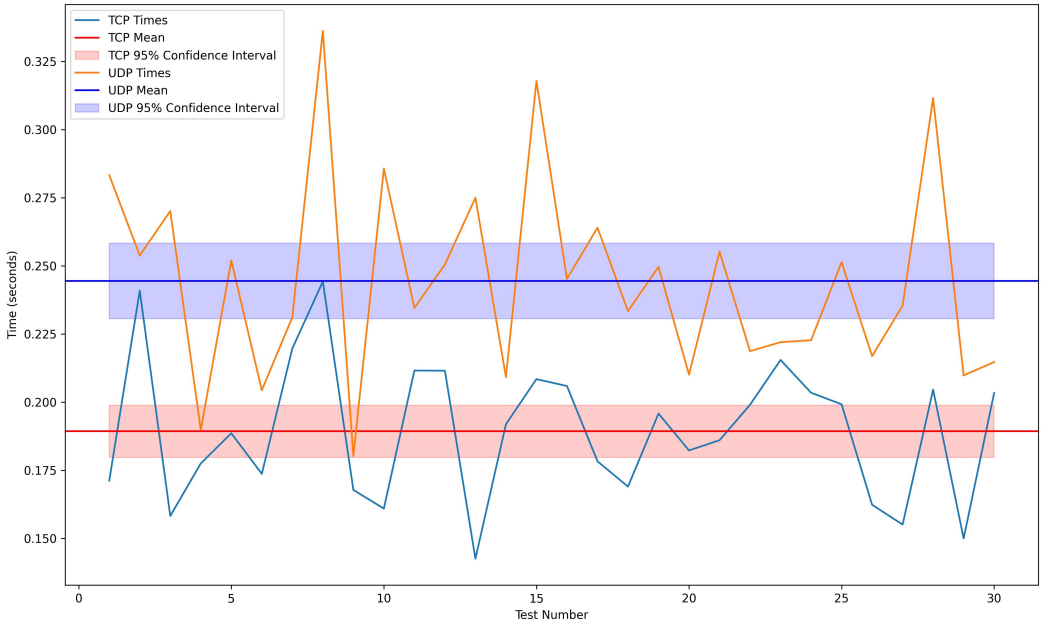



Figure 1: Benchmark test.

## 4.2 Packet Loss Testing

We test our implementations with packet loss testing because in a real world scenario, not every packet gets delivered where it supposed to. So a good protocol must be able to handle the lost packet and while doing so its speed must not be effected.

### 4.2.1 %5 Packet Loss

In the %5 packet loss test we have observed that TCPs speed have greatly decreased, while our UDP implementation was slowed down only a little. That is because UDP have both RDT3.0 and selective repeat algorithm implemented in it. Also UDP does not have a handshake protocol and no connection.

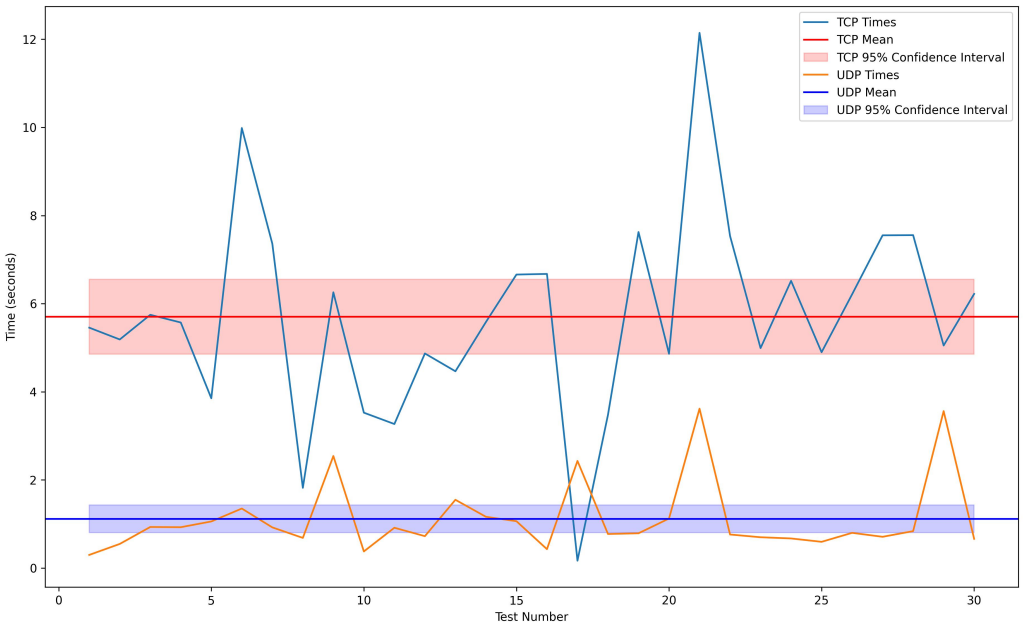



Figure 2: %5 Packet loss test.

### 4.2.2 %10 Packet Loss

It seems that because of TCPs ACK handling logic, when packet loss increases, TCP slows down a lot. Because of that, again our UDP implementation works much better than TCP with %10 packet loss.

Finding optimal packet size for TCP is difficult, because smaller packet sizes like 512, 1024 takes a lot of time to transmit while bigger packet sizes increase the chance of corruption while sending. Our UDP implementation does not seem to suffer from this matter because it ensures pipelined connection for faster transmission rates for smaller packets.

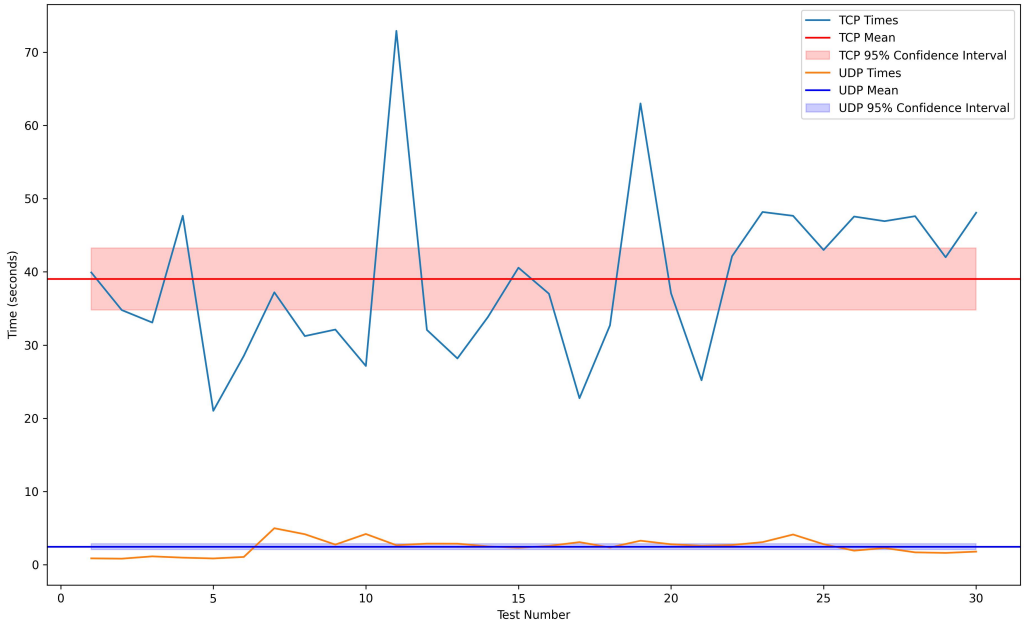



Figure 3: %10 Packet loss test.

### 4.2.3 %15 Packet Loss

At this point, a %5 increase in packet loss causes nearly quadratic increase in transmission times of TCP packets. Middle-large sized packets tend to be more corrupted in this scenario; hence our TCP implementation becomes much slower. On the other hand, our UDP socket transmits smaller sized packets more securely while providing enhanced speed of selective repeat implementation.

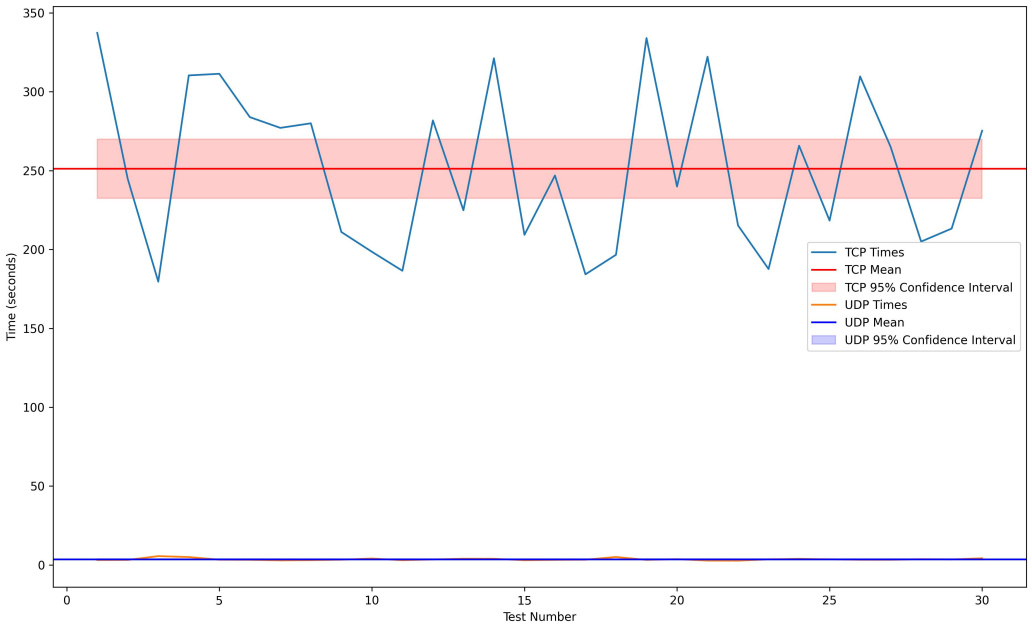



Figure 4: %15 Packet loss test.

## 4.3 Packet Corruption Testing

Testing our implementations in environments where some packets get corrupted is crucial, because all packets may not get delivered in the structure they supposed to be in. Corrupted packets must be re-transmitted to the receiver seamlessly and this operation should not cause significant downtime on our sockets.

### 4.3.1 %5 Packet Corruption

As you can see from the graph below, a %5 packet corruption rate causes TCP to fluctuate between 4 and 8 seconds with an average of nearly 6 seconds. 6 seconds is a significant amount of time in the Internet environment, and repetitions of this process can be costly. However, our UDP implementation solves this problem and keep the downtime at an average below 1 second. This provides seamless re-transmission of corrupted packets and enhances user experience.

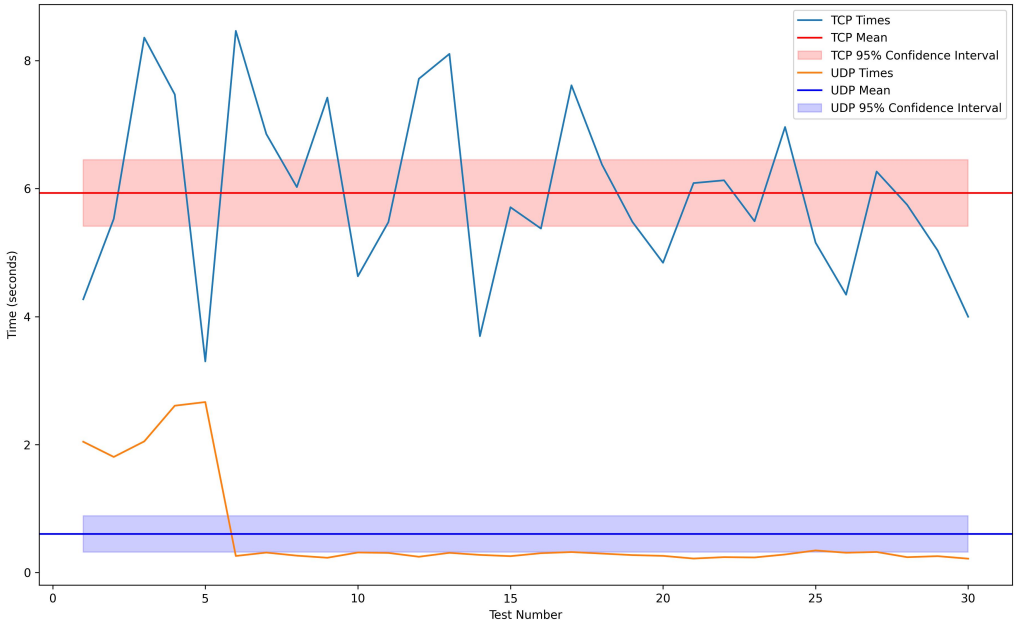



Figure 5: %5 Packet corruption test.

### 4.3.2 %10 Packet Corruption

In this test scenario, it is more evident that TCP can sometimes be very disadvantageous against UDP because there is a huge time difference for transmission of the packets. TCP can see a downtime of 35 seconds while UDP maintains its transmission rate with a constant time of nearly 0.2 seconds.

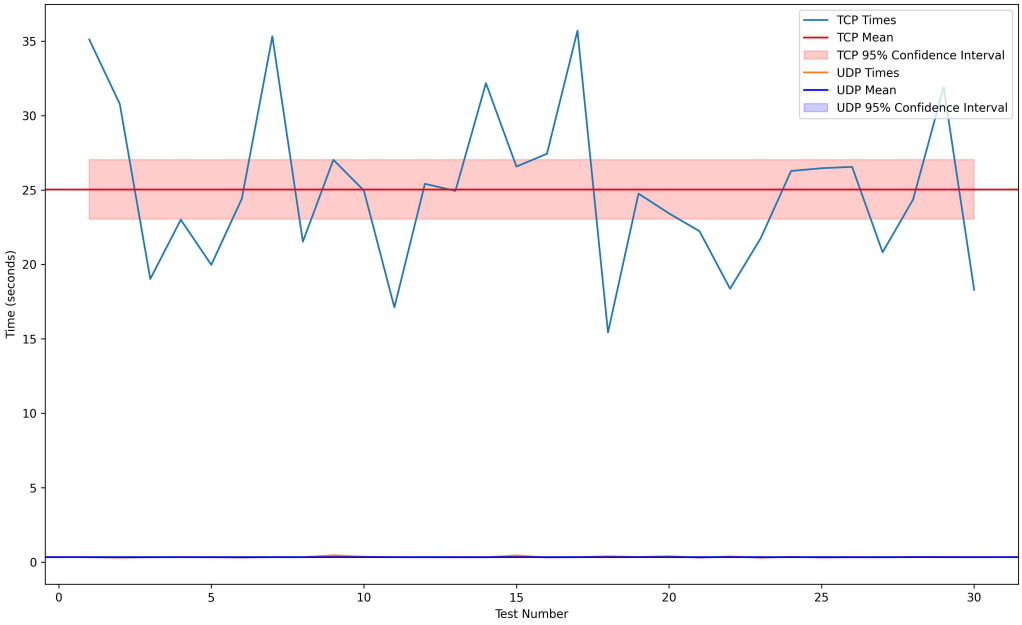



Figure 6: %10 Packet corruption test.

## 4.4 Packet Duplication Testing

We implement this kind of testing because in a real world scenario, sometimes ack's that the receiver sends to the server may be lost or corrupt on the way. This causes receiver to receive duplicate packets. It may cause downtime on both server and client sides.

### 4.4.1 %5 Packet Duplication

Our TCP implementation's duplicate packet handling seem to be slower than our UDP implementation's. The main reason for that may be our UDP implementation ignores duplicates, because when the UDP client gets the packet, it remembers its sequence number and does not include same sequence number in further calculations.

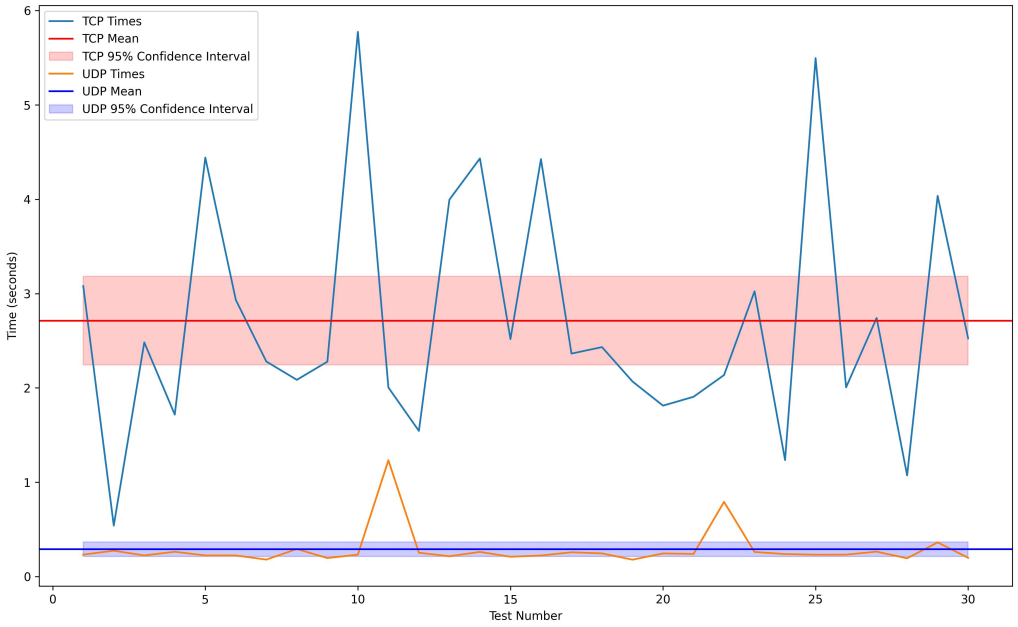



Figure 7: %5 Packet duplication test.

### 4.4.2 %10 Packet Duplication

Again, when packet duplication increases in this scenario, our TCP implementation seem to be more affected by this; however, this test shows that in the UDP implementation, there can be some unexpected fluctuations that may cause unwanted delays. But overall, UDP seems to be much more faster than TCP in this case.

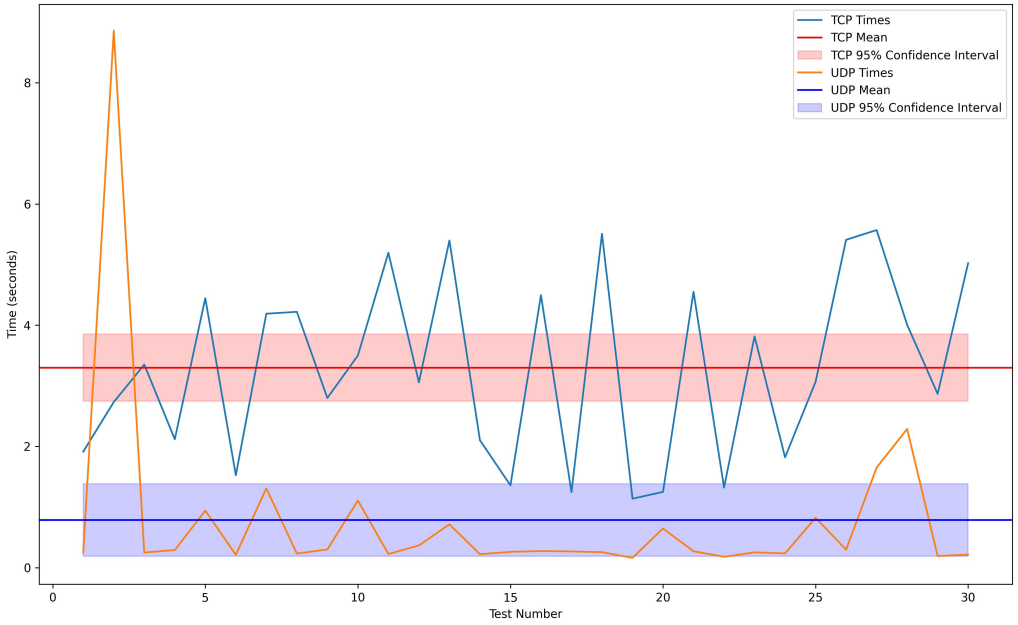



Figure 8: %10 Packet duplication test.

## 4.5 Packet Delay Testing

We do this kind of testing because in a real world scenario, there can be a traffic. This means your data can arrive later than it should. We tested for two scenarios for delay. First one is normal delay, the one you can experience in real life, the other one is the uniform delay, all packets are hold a certain time before it gets delivered.

### 4.5.1 Normal Delay

Our UDP implementation seems to handle normal delay better than TCP implementation. This shows that our UDP implementation is a better fit for real life using than our TCP implementation.

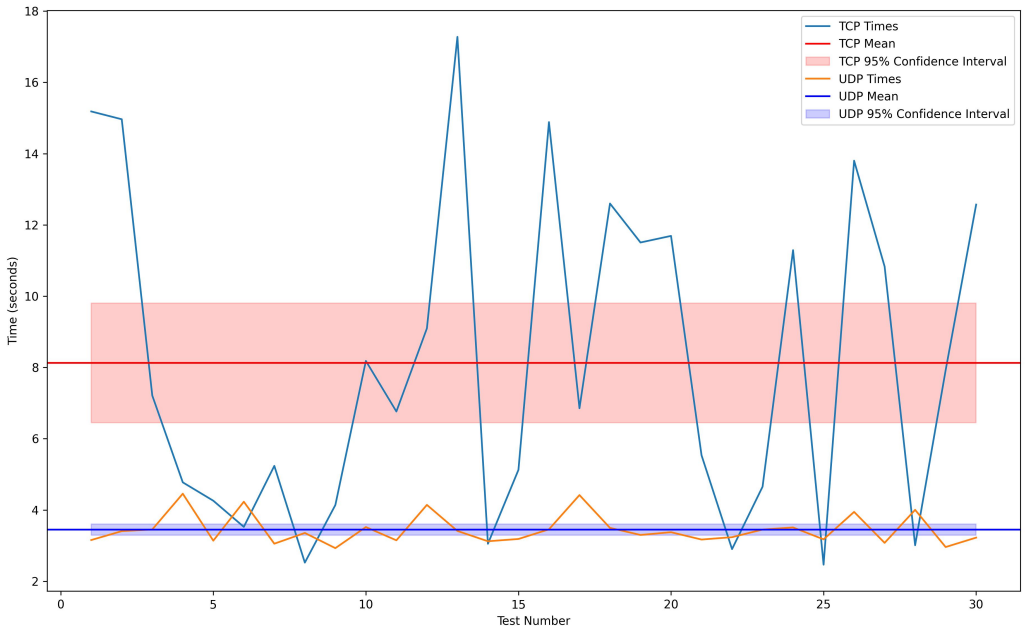



Figure 9: Normal delay test.

### 4.5.2 Uniform Delay

Unlike normal delay, our UDP implementation could not handle uniform delay as much as our TCP implementation. The main reason for that may be that there is no connection between the server and client in the UDP implementation and that may cause some ACKs or packets get lost on the way because of the delay.

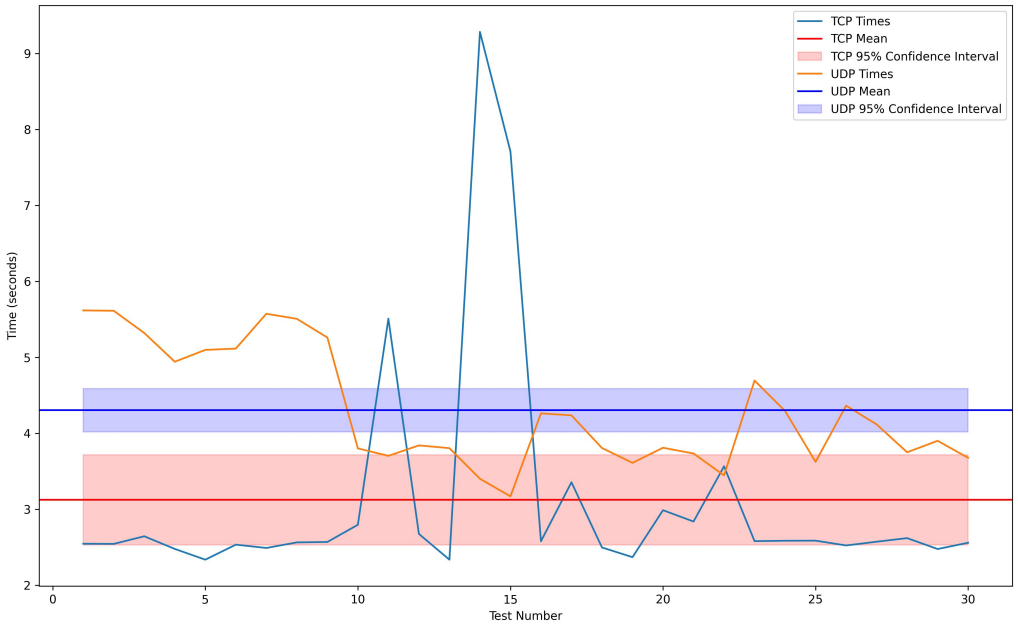



Figure 10: Uniform delay test.

## 4.6 Final Notes

Overall our UDP implementation seems to be working really well. It couldn't pass our TCP implementation on some cases but that doesn't mean that we cant use our UDP implementation in a real life scenario. Our UDP implementation couldn't pass the TCP implementation on the benchmark and uniform delay tests, but those tests are the most far away scenarios from real life.