UDP Design

Our design consists of 3 protocol layers on top of each other. We first describe the first layer (bottom-up).

RDT

This layer will do segment transfer. It takes as input stream of segments where each segment is less than 2.5 KB. It sends segments with Selective-Repeat pipelining and there is no ordering when sending. There is also no sliding window in our design.
Each segment has an ID number.
Our protocol guarantees the reliable data transfer of the segments. We calculate checksum for our defined headers and the payload in this protocol to make sure there are no corrupt packets.

How does it work?

We have two threads:

1. Sending thread: This thread takes a packet from the sending buffer and appends it to the wait_for_ack buffer. Then it sends it. It will keep sending until the size of the wait_for_ack buffer equals window size.
2. Receiving thread: This thread receives data packets and appends them to the received buffer. After that it will send ACK for the received packet. In this layer duplicate packets are not handled and deliver such packets to the receiving end. This case is handled in the upper layer protocol which we'll explain.

Initialization step

- Server side: In this step, it just starts the sending and receiving threads and keeps running.
- Client side:
    - First it starts its own sending and receiving threads.
    - Sends an "i" packet to the server to start connection and creates a timer for it. Then the sending thread will sleep.
    - When the receiving thread receives a "s" packet, it cancels the timer and wakes up the sending thread.
    - In this case the connection has started.

Transfer step

- Server side:
    - Takes a message less than 2.5 KB and adds header that contains:
        - (type "d", segment ID, Length, Payload)
    - Then it sends this packet using UDP to the other side and starts a timer for it.
    - Afterwards, it appends packets to the wait_for_ack buffer.
    - Keeps sending packets until the window size limit is reached.

    - Now when a packet is sent, we have 2 cases: either it receives ACK for it or timeout occurs.
        - Case Receive ACK -> Fetches the next segment and removes from buffer.
        - Case Timeout: Resends and restarts the timer.
    - The receiving thread will receive ACKS and deliver to the higher layer protocol.

Closing step:

- Server side:
    - It first receives a “c” packet for closing the connection.
    - Blocks adding more data to the sending buffer.
    - If the sending buffer is empty, it will stop sending packets.
    - When the wait_for_ack buffer is empty, it ACKs closing message.
- Client side:
    - Client blocks sending more messages, waits for the buffer to empty and kills sending thread.
    - Sends “c” packet for closing connection to server, starts a timer for it and sleeps thread.
    - After the client receives ACK for the “c” packet and wait_for_ack is empty, kills the receiving thread and closes the connection.

Interleaving Protocol (RDT+)

Sender side:
In the next layer, we do interleaving. It is defined in the `rdtplus.py` file. Essentially it has a single function called send that takes the list of messages from the application layer and the address to send them.
It splits messages into 2KB segments, interleaves them and sends them to the RDT protocol below.
It also adds the following headers:

(Object no., Seg no., Max segment number)

Receiver side:
The receiver side of this protocol receives the segments and if they are duplicated, it discards them. When the object is complete (all segments of an object are received), it delivers to the higher layer protocol.

Application Layer Protocol

This protocol is simply as follows:

1. Send a start transfer message.
2. Sender sends a total number of files to transfer.
3. Receiver sends “get” to get the files.
4. Sender sends files through RDT+ protocol.

5. When the receiver gets all the files, it sends an “OK” message and closes the connection.

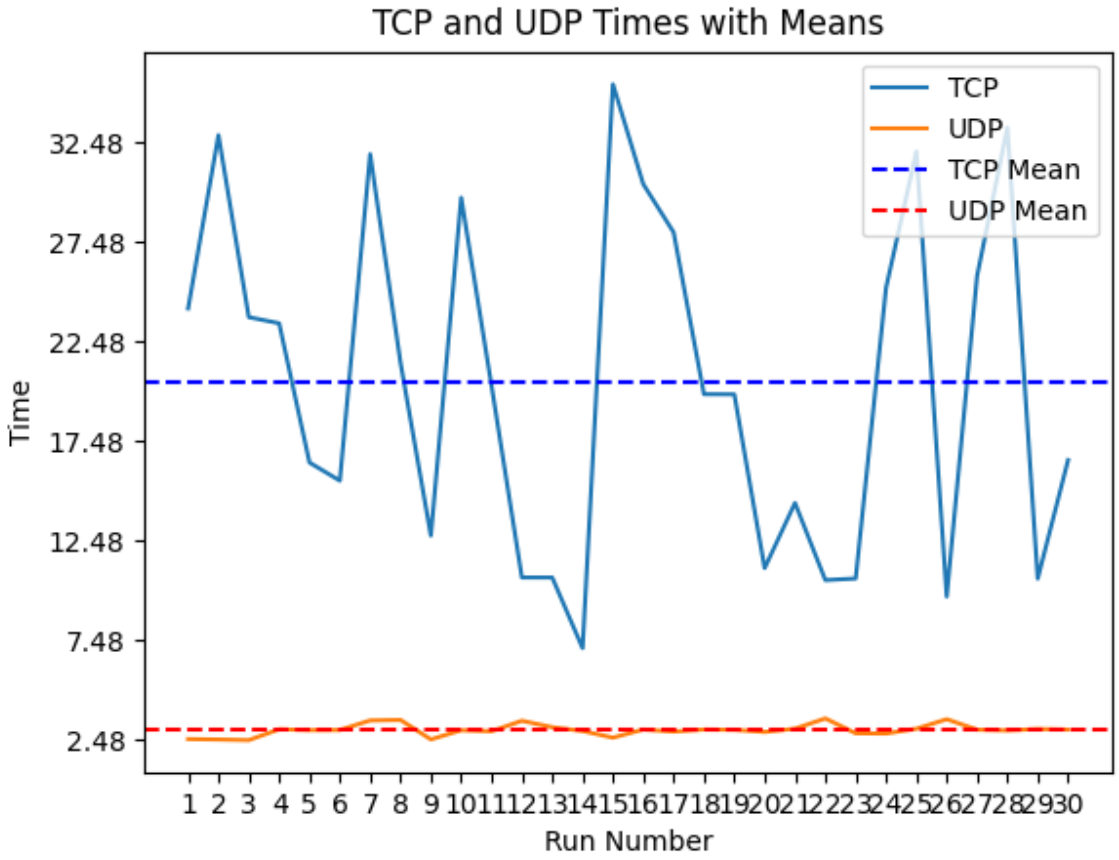
Experiments

We conducted experiments with different parameters 30 times and collected their mean, confidence interval (95%) and plotted their graphs. All the graphs in the following sections are:Time (s) vs Run number graphs.

The table below will depict the means and confidence intervals of the experiments:

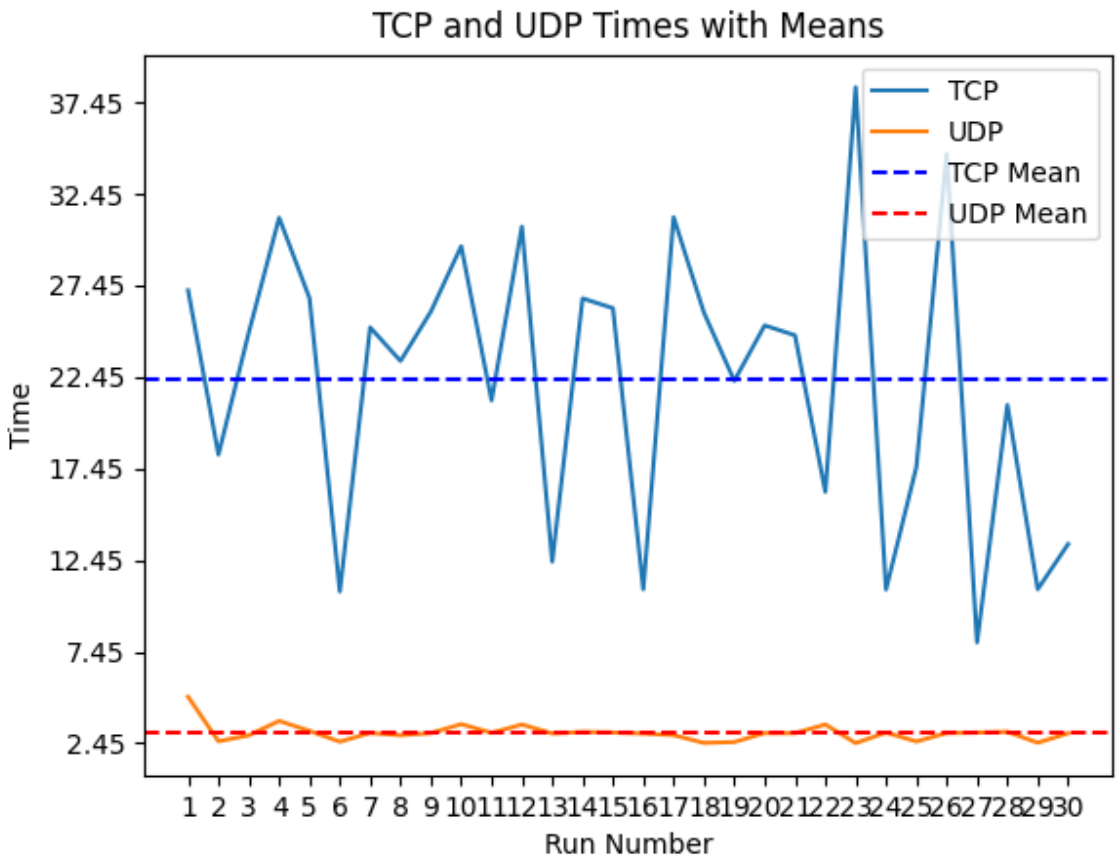
Experiment	TCP Mean (s)	TCP Confidence Interval	UDP Mean (s)	UDP Confidence Interval
baseline	20.44	(17.22, 23.65)	2.99	(2.88, 3.10)
loss_0	22.36	(19.41, 25.32)	3.01	(2.83, 3.20)
loss_5	26.59	(23.44, 29.74)	4.5	(3.73, 5.27)
loss_10	33.09	(30.13, 36.04)	8.26	(5.06, 11.46)
loss_15	260.28	(167.56, 352.99)	43.65	(40.93, 46.37)
corrupt_0	24.31	(21.58, 27.04)	2.91	(2.76, 3.05)
corrupt_5	30.06	(24.03, 36.09)	5.04	(3.42, 6.67)
corrupt_10	33.1	(30.86, 35.34)	5.65	(5.27, 6.03)
duplicate_0	24.7	(21.90, 27.50)	2.95	(2.85, 3.04)
duplicate_5	28.29	(25.00, 31.59)	2.98	(2.77, 3.18)
duplicate_10	30.66	(27.79, 33.52)	3.79	(2.94, 4.65)
delay_100ms_ uniform	26.57	(25.73, 27.40)	3.17	(3.14, 3.20)
delay_100ms_ normal	30.86	(29.00, 32.71)	4.15	(4.04, 4.26)

Baseline

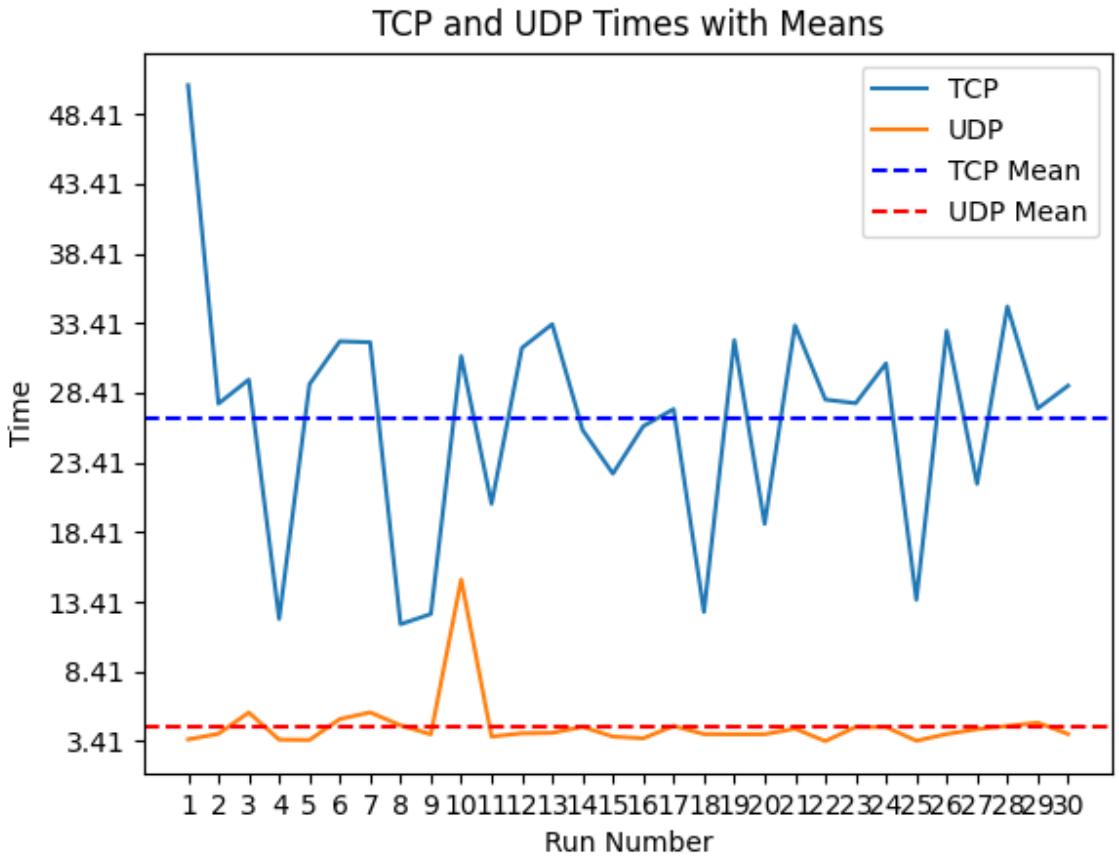

TCP and UDP Times with Means
TCP
UDP
TCP Mean
UDP Mean
Time
32.48
27.48
22.48
17.48
12.48
7.48
2.48
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
Run Number

Packet Loss

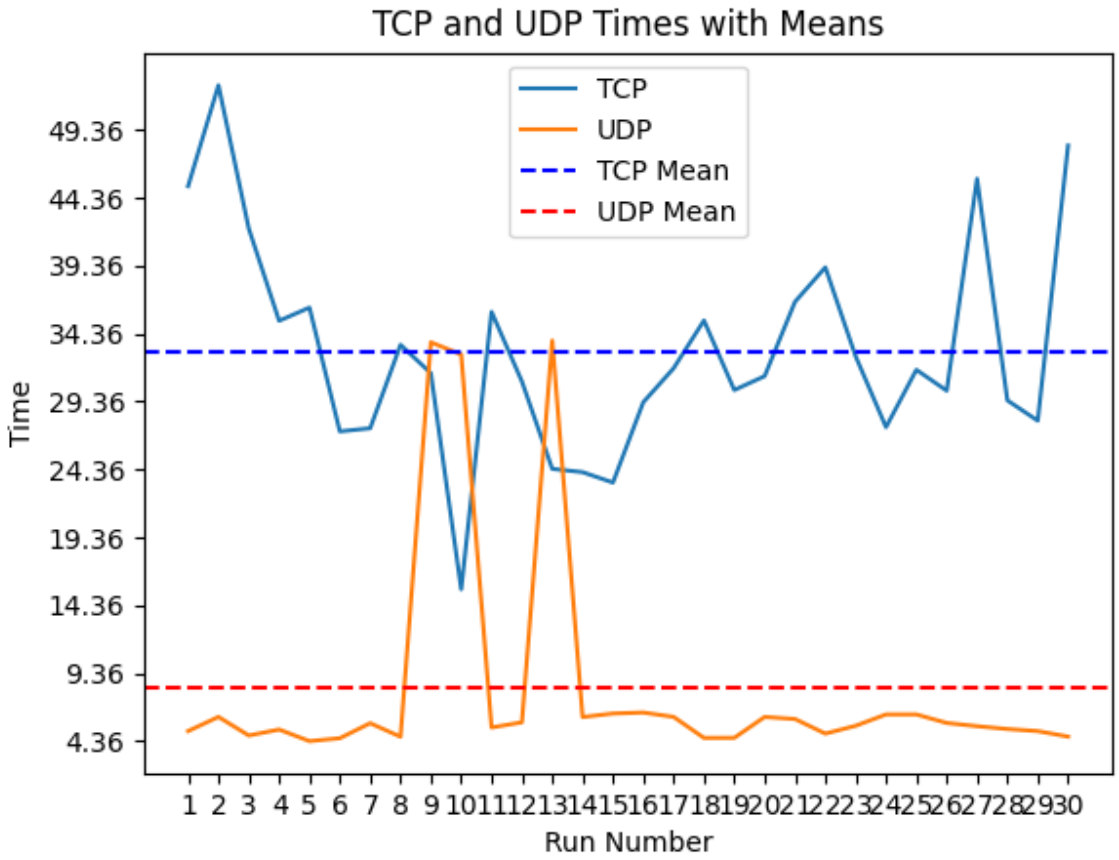
Loss 0%



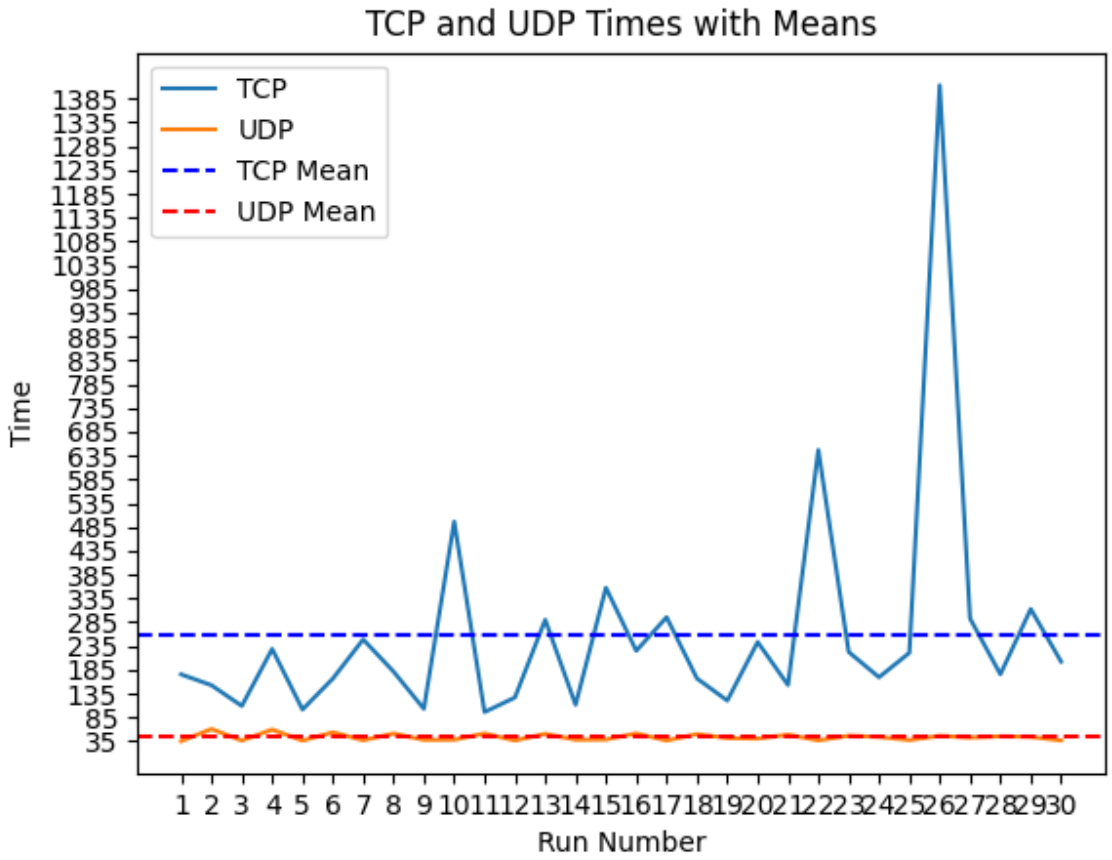
Loss 5%


TCP and UDP Times with Means
TCP
UDP
TCP Mean
UDP Mean
Time
48.41
43.41
38.41
33.41
28.41
23.41
18.41
13.41
8.41
3.41
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
Run Number

Loss 10%

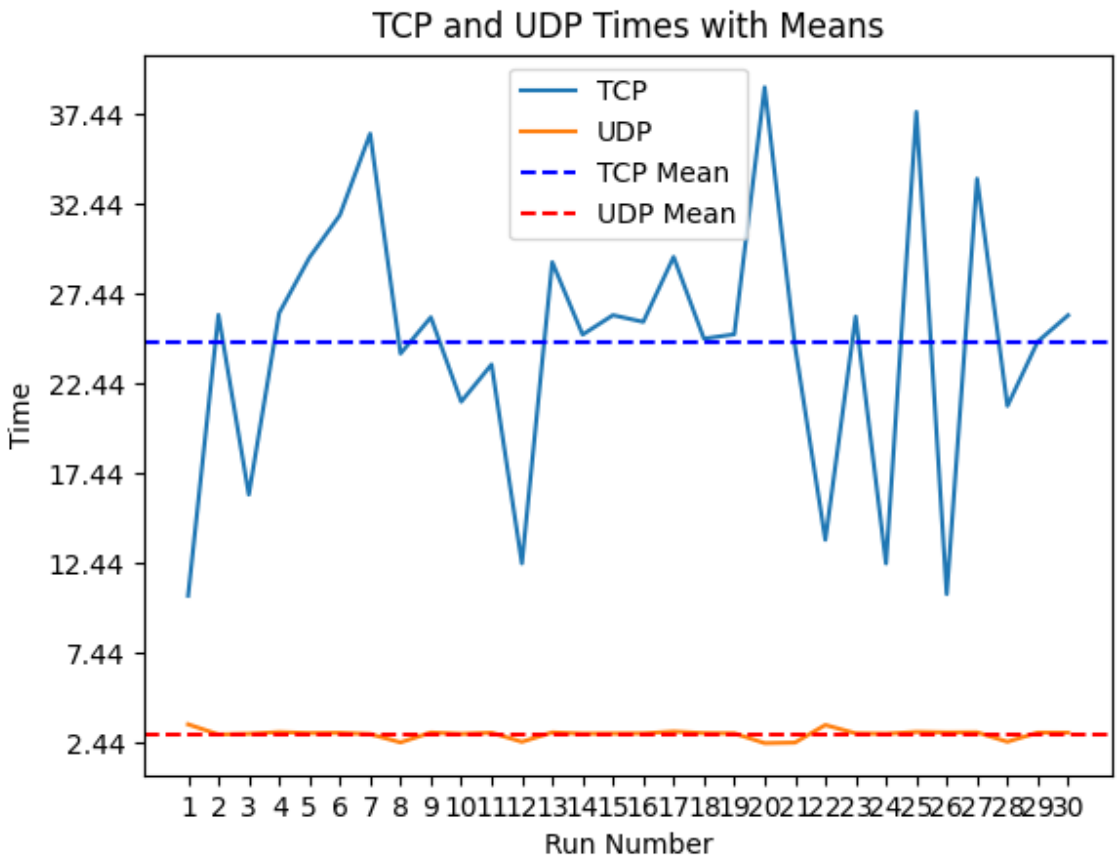

TCP and UDP Times with Means
TCP
UDP
TCP Mean
UDP Mean
Time
49.36
44.36
39.36
34.36
29.36
24.36
19.36
14.36
9.36
4.36
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
Run Number

Loss 15%

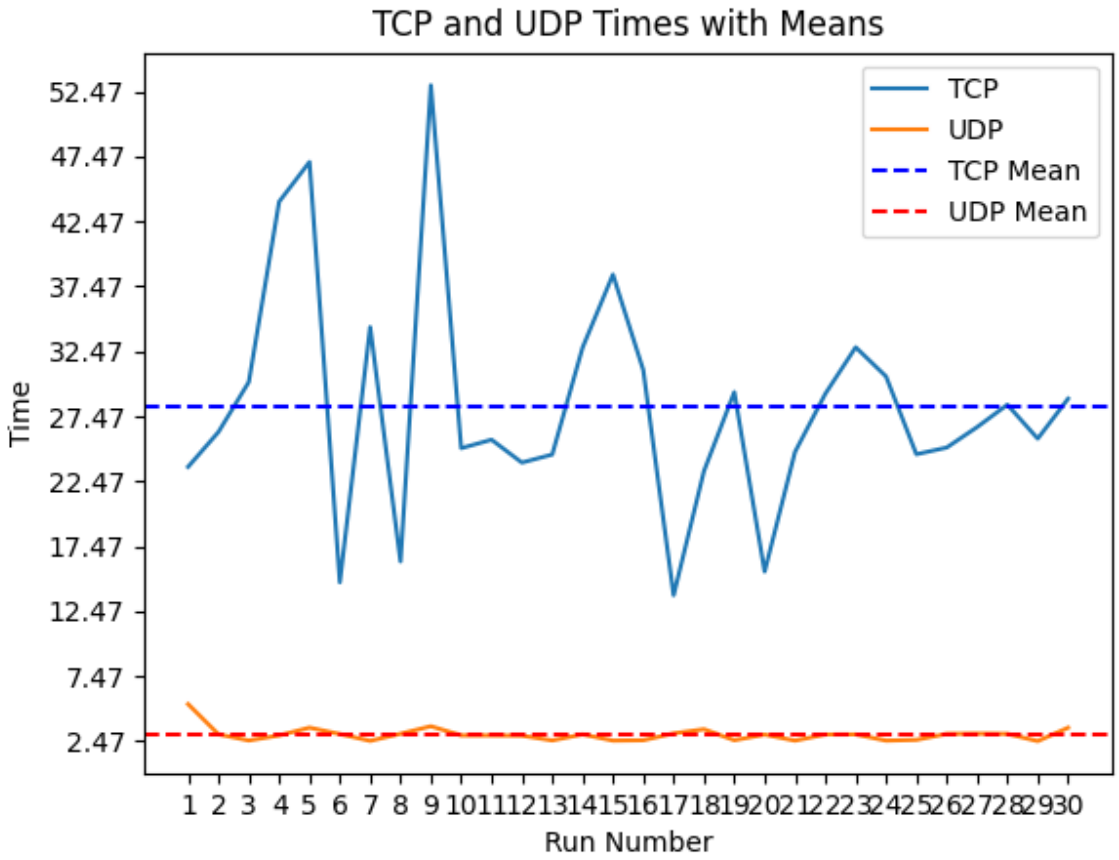

TCP and UDP Times with Means
TCP
UDP
TCP Mean
UDP Mean
Time
Run Number

Packet Duplication

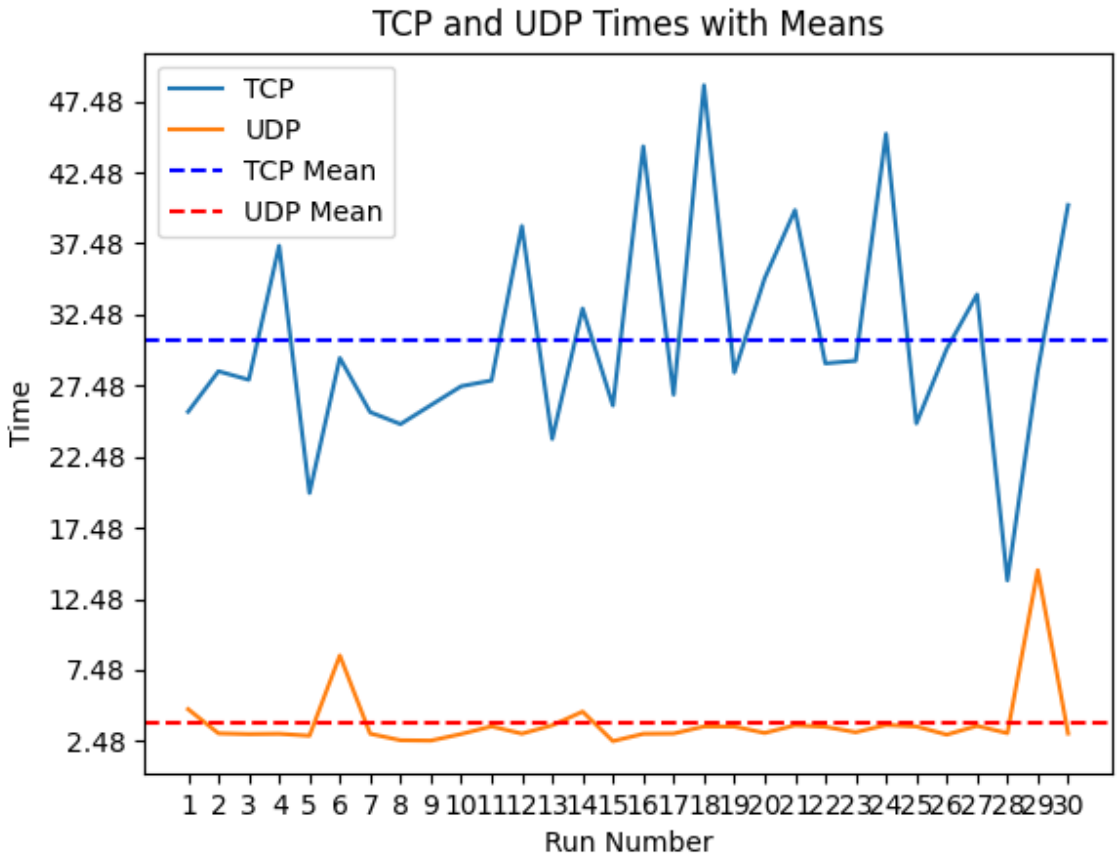
Duplication 0%



Duplication 5%

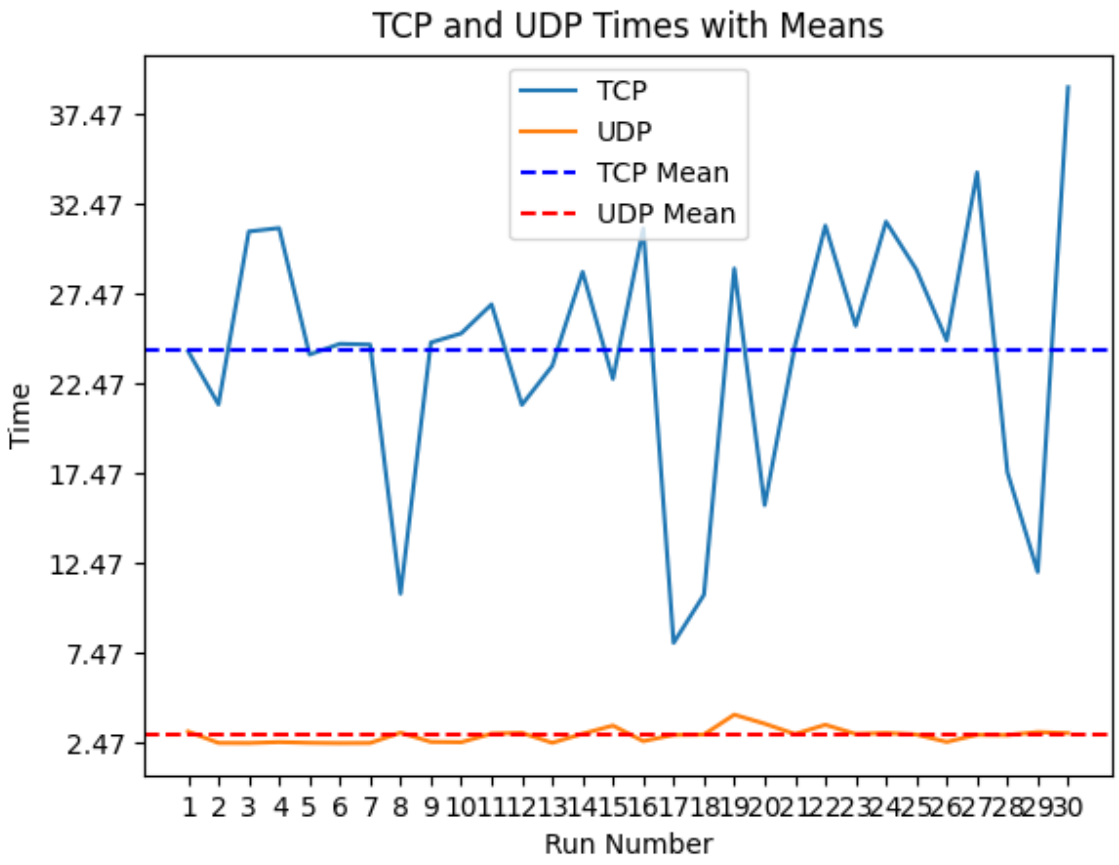


Duplication 10%

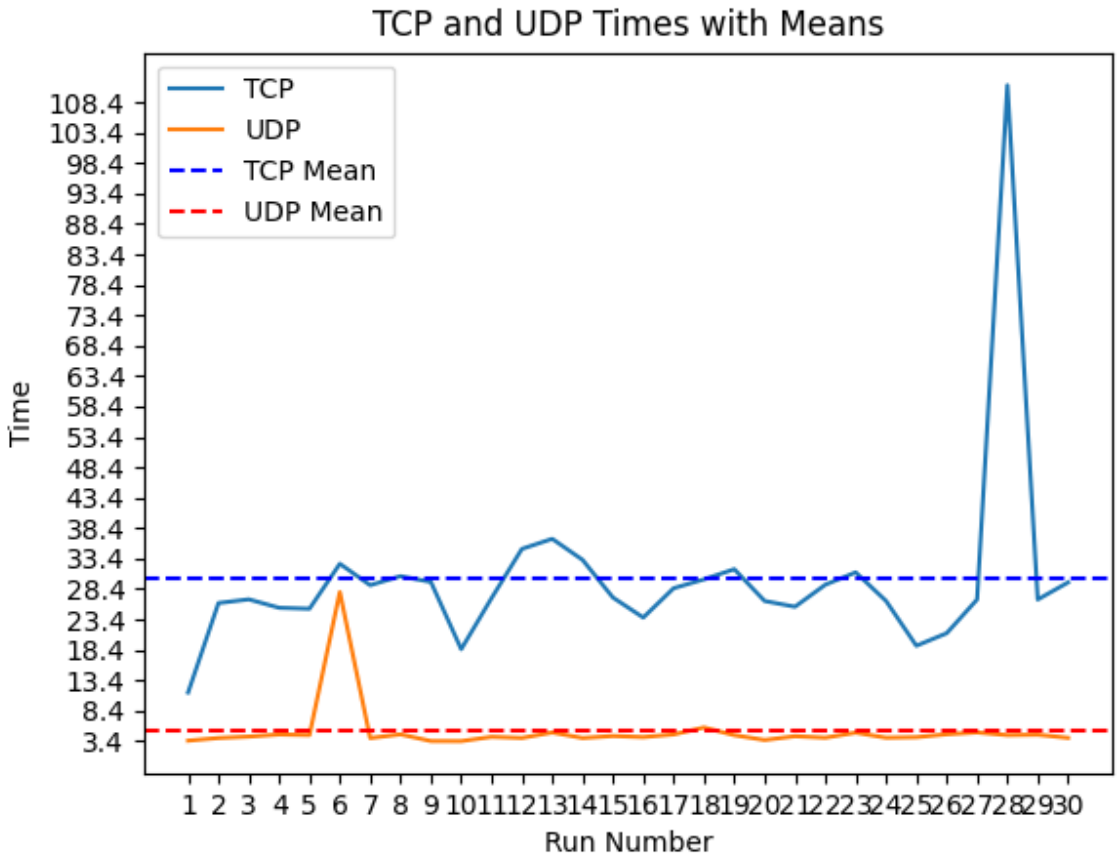


Packet Corruption

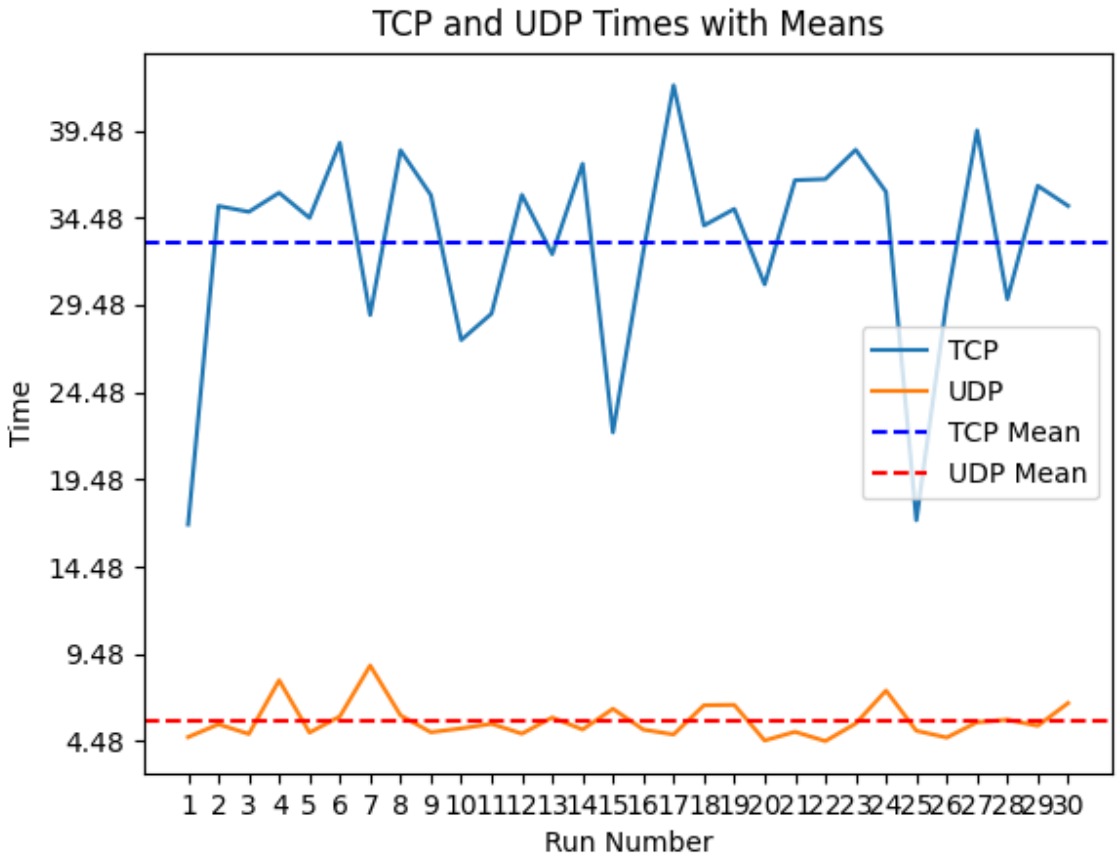
Corruption 0%



Corruption 5%

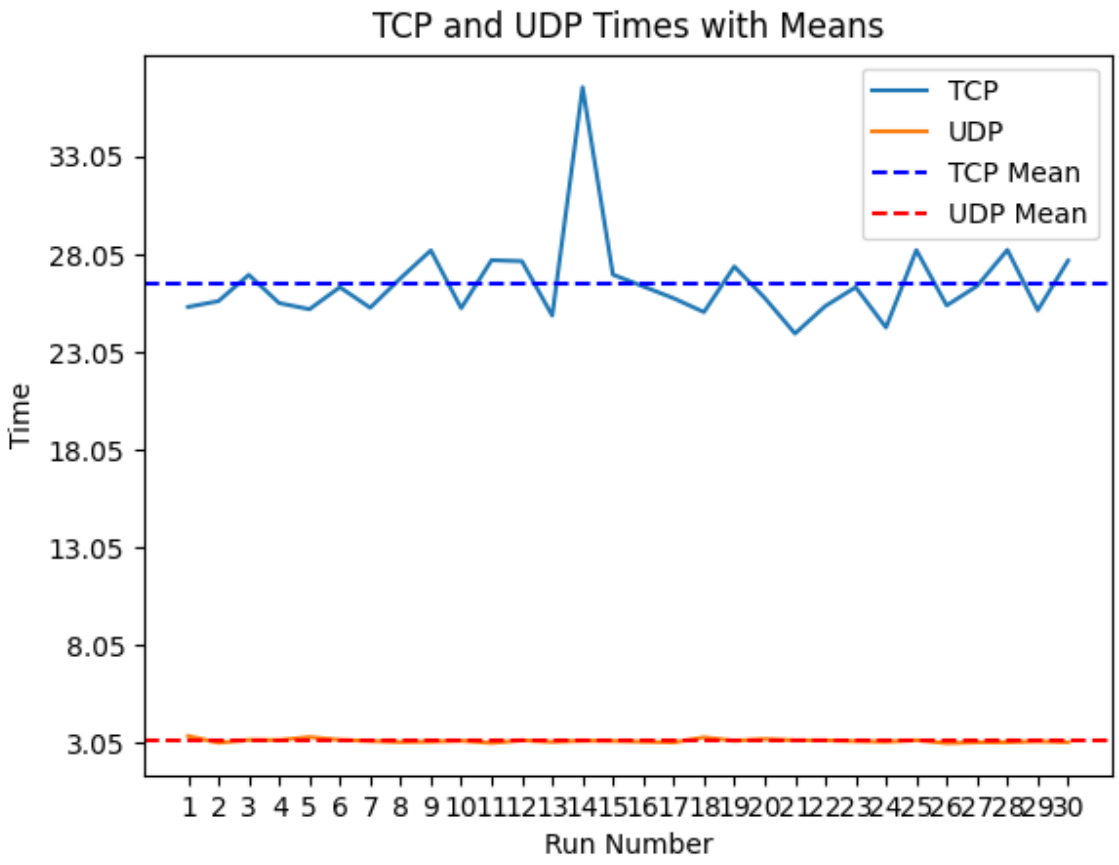


Corruption 10%

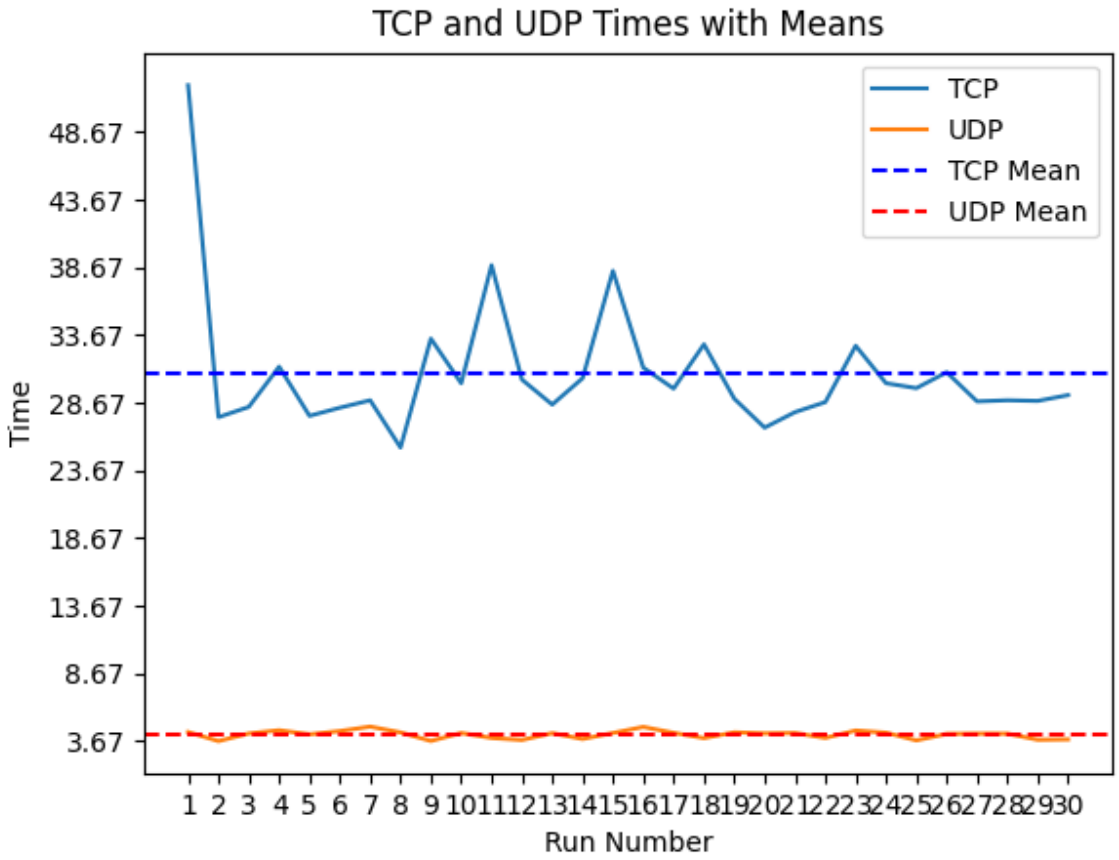


Packet Delay

100ms Uniform Distribution



100ms Normal Distribution



Analysis of Results

Based on our experiments, our implementation of reliable data transfer (RDT & RDT+) is faster than TCP in almost all our experiments. The reason for this performance increase is due to the following reasons:
1. Multithreading
2. No HOL blocking. One important reason for the good performance is the fact that we do interleaving in the application layer, therefore it boosts the file transfer.
2. No congestion control. We just guarantee data is transferred but unlike TCP there is no mechanism for congestion control. Therefore TCP is slowed down for this mechanism.

In our experiments, with WINDOW_SIZE = 1000 and SEGMENT_SIZE = 2048, we were able to outperform TCP. However these values didn't manage to outperform TCP in the packet delay from normal distribution with value 100ms (20ms jitter value). We experimented again by adjusting SEGMENT_SIZE to 1024 bytes, and it was able to outperform TCP this time as seen in the graph and table.