CENG 435 - Data Communications and Networking
Fall 2023
Programming Assignment

Anıl Eren GÖÇER
e2448397@ceng.metu.edu.tr

Furkan KÖROĞLU
e2448645@ceng.metu.edu.tr

January 3, 2024

Introduction

In this report, we will propose a novel reliable file (object) transfer protocol over UDP and inspect its relative performance compared to TCP via experiments. We have conducted 13 different experiments in different network (referred as "link" in the homework specification) states and we will examine each case one by one and try to explain why we have obtained such results. In addition to these, we will briefly introduce our approach (algorithm) before the experiments.

Note: We have assumed that the files (object) will have the same size as the files (objects) given to us. Therefore, please keep in mind this fact while you are trying the code.

Note: We have fine-tuned our code during the experiments in a static way. Therefore, please follow the specifications regarding command line arguments given in the README file in order to obtain the same results.

Note: We will refer the novel reliable file (object) transfer protocol over UDP simply as UDP in the graphical comparison plots and numerical figures in the Appendix.

Note: Since some of the representations are programming language specific, they are not mentioned in Algorithm 1 and 2. If you want to grasp all details, examine these algorithms together with the code.

Note: All the time values in the experiments are given in seconds.

Algorithm

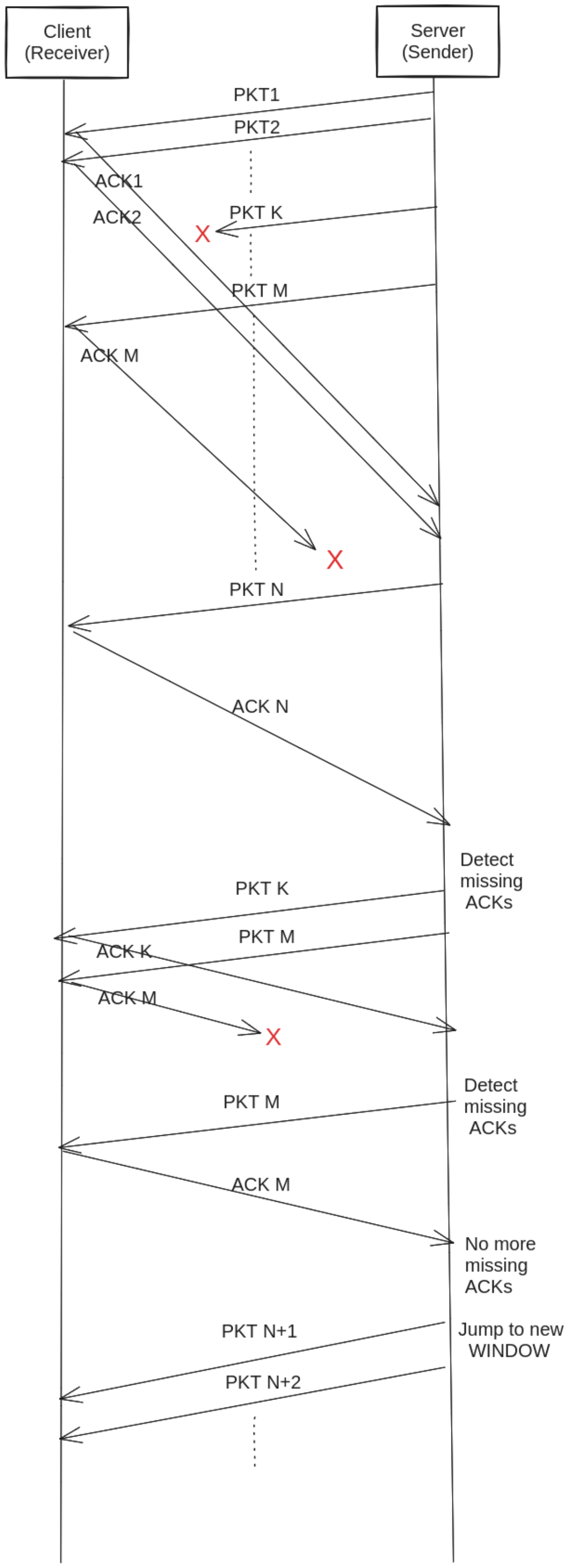

Figure 1: Demonstration of the Reliable File (Object) Transfer Protocol over UDP

Our proposal for reliable object transfer protocol over UDP works as described in Figure 1.

From the perspective of receiver (client in this case), it just wait for incoming packets. It checks if they are corrupted, and if they are not, then it process these packets accordingly. Packets transmitted from server to client are named as **Eventual Packets**, it is like a wrapper over a **Pure Packet**. Their organization is described as follows.

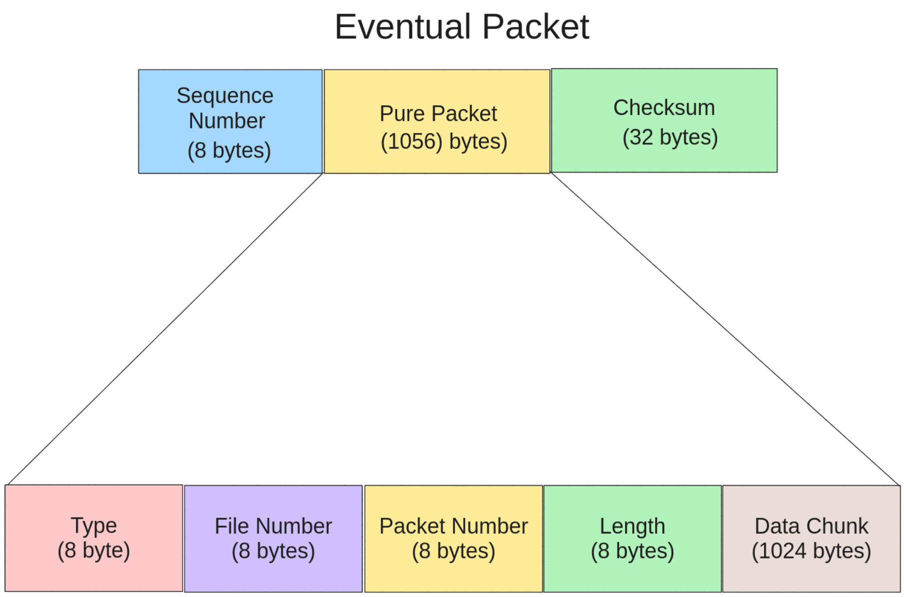

Figure 2: Demonstration of the Reliable File (Object) Transfer Protocol over UDP

For an Eventual Packet:

- **Sequence Number:** It shows the location of an eventual packet in the all eventual packets transferred between client and server. It takes the values in the interval $[1, EVENTUAL\_PACKET\_COUNT]$.
- **Pure Packet:** It is just an abstraction over a collection of header specifying a data part in an object and data itself (named as data chunk).
- **Checksum:** It is a checksum calculated over sequence number and pure packet to detect corruption of the eventual packets.

For a Pure Packet:

- **Type:** It specifies if the data chunk belongs to large file or a small file. If it is 0, then it means this data chunk is coming from a small object. If it is 1, then it belongs to a large object.
- **File Number:** It represents the number of file being transmitted. It takes the values between 0 and 9 inclusively.
- **Packet Number:** It represent which pure packet is being transmitted belonging to a file (object). If an object consists of C data chunks, then this packet number header takes values in the interval $[1, C]$.

- **Length:** Some of the data chunks are padded with 0 at the end for packets to have an uniform size. Specifically, these packets are the last packets belonging to each object (file). In order to process these packets correctly at the client side, this value specifies the size of the chunk without padding.
- **Data Chunk**: It is the data itself read from the object (file).

Client (receiver) receives eventual packets from the server (sender) and checks if it is corrupted or not. If it is not corrupted, it extracts the pure packet. If the eventual packet has the sequence number 9999, it is not processed because sequence number 9999 is reserved for representing the termination packet. For all the other packets with any sequence number, pure packet is processed with respect to header values in the pure packet and corresponding data chunk is written to matching file. Lastly, it sends an Acknowledgment packet with the number same as sequence number of the eventual packet.

From the perspective of the server (sender), it is responsible for sending files by dividing them into smaller parts. It has a parameter to be fine-tuned which is $WINDOW\_SIZE$. It divides all the eventual packets into groups of size $WINDOW\_SIZE$. For each window, server sends all the eventual packets in the windows quickly (without waiting for an acknowledgement individually). Then, it reads incoming acknowledgements (possibly with missing ones). After this step, it detects missing acknowledgments and resends those packets until there is no remaining missing acknowledgement. Once the server collects all the acknowledgements corresponding to the window, it **jumps** to the next window. The keyword **jump** is chosen instead of slide intentionally because the window does not slide by 1 rather it slides by the amount of $WINDOW\_SIZE$. It repeats this process until all the eventual packets are sent and acknowledged.

Formal algorithms for the client (receiver) and server (sender) are given as follows:

Algorithm 1 Receiver (Client)

```
while True do
    eventual_packet ← recieve_from_server()
    sequence_number ← get_sequence_number(eventual_packet)
    pure_packet ← extract_pure_
    if sequence_number == termination_code then
        break
    else
        if eventual_packet is corrupted then
            continue
        else
            send_acknowledgement_to_server(sequence_number)
            write_chunk_to_corresponding_file(pure_packet)
        end if
    end if
end while
```

Algorithm 2 Sender (Server)

Require: $EVENTUAL\_PACKET\_COUNT \ \% \ WINDOW\_SIZE = 0$
  while $WINDOW\_BASE$!= $EVENTUAL\_PACKET\_COUNT$ **do**
    for *packet* with *packet_number* in $[WINDOW\_BASE, WINDOW\_END]$ **do**
      $send\_to\_server(packet)$
    end for
    $received\_acks \leftarrow receive\_acks\_from\_client()$
    $missing\_acks \leftarrow detect\_missing\_acks(WINDOW\_BASE, WINDOW\_END)$
    while *missing_acks* is **not** *empty* **do**
      for *missing_ack_number* in *missing_acks* **do**
        $send\_to\_server(missing\_ack\_number)$
      end for
      $missing\_acks \leftarrow detect\_missing\_acks(WINDOW\_BASE, WINDOW\_END)$
    end while
    $WINDOW\_BASE \leftarrow WINDOW\_BASE + WINDOW\_SIZE$
    $WINDOW\_END \leftarrow WINDOW\_BASE + END\_SIZE$
  end while

Experiments

Nearly Perfect Network State

In this subsection, we will analyze the cases in which there is no netem rules applied, or Packet Loss with the probability 0%, or Packet Duplication with the probability 0%, or Packet Corruption: with the probability 0%. All these states actually belonging to the same case, so we decided inspect them together.

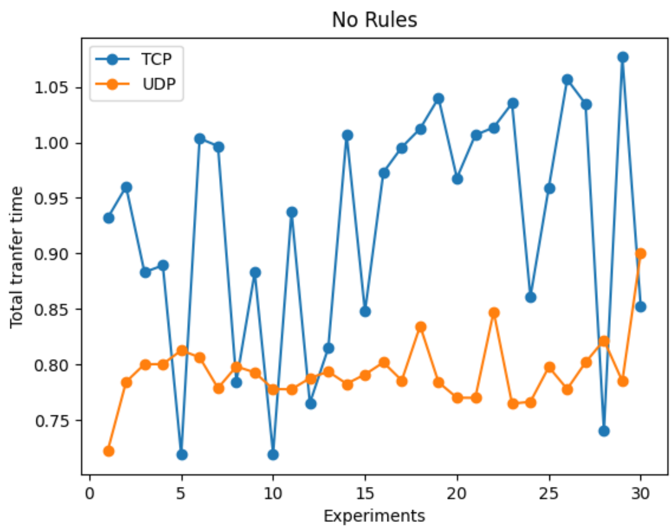

Figure 3: Total Transfer Time for TCP vs UDP - no netem rules applied

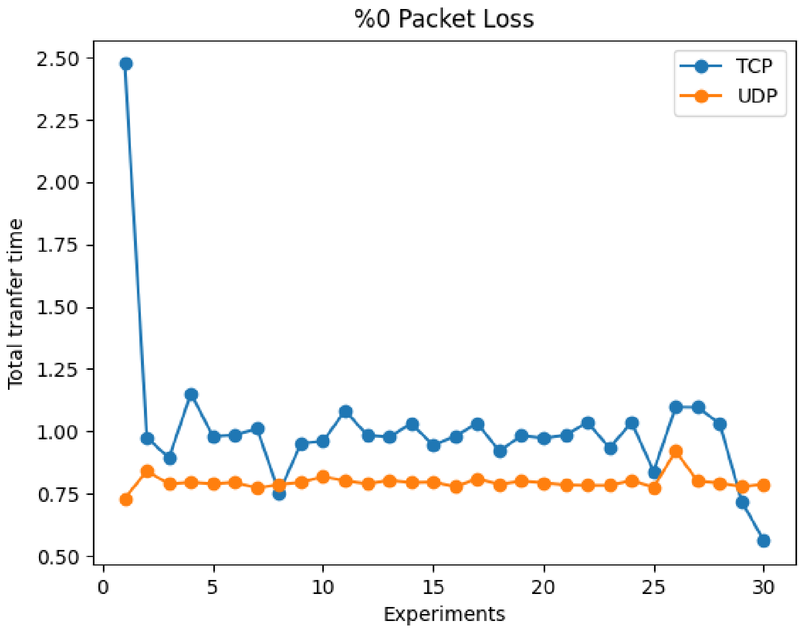

Figure 4: Total Transfer Time for TCP vs UDP - packet loss probability is 0%

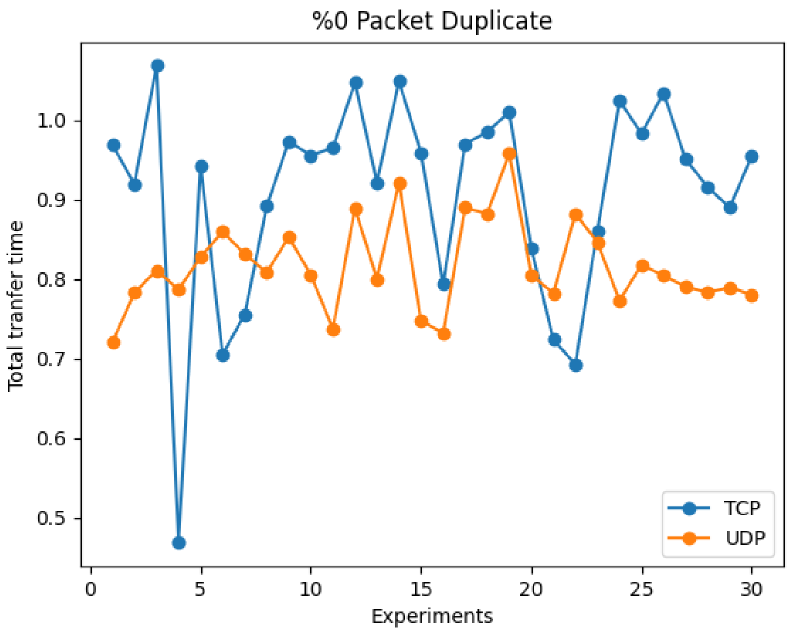

Figure 5: Total Transfer Time for TCP vs UDP - packet duplication probability is 0%

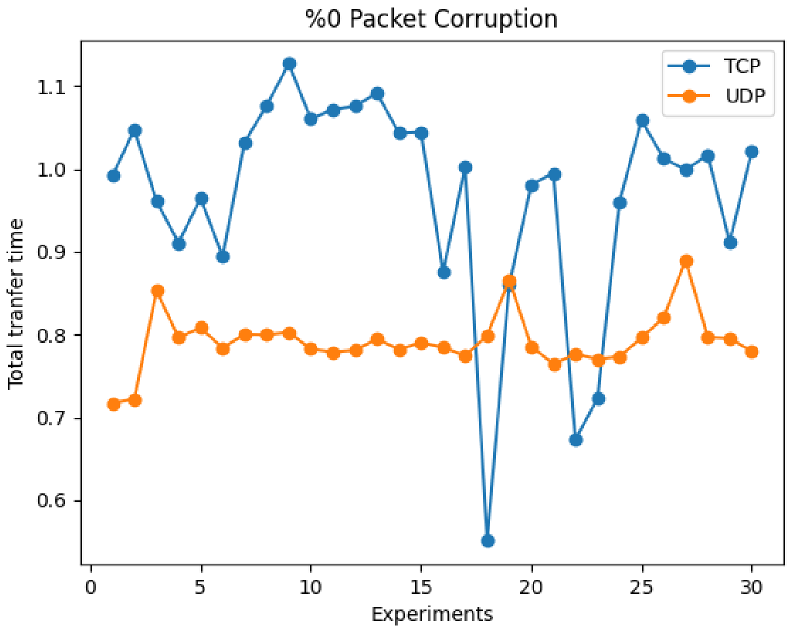

Figure 6: Total Transfer Time for TCP vs UDP - packet corruption probability is 0%

As it can be seen from the Figure 3, 4, 5 and 6, our object (file) transfer protocol over UDP performs better (faster) than TCP. First of all, we need to emphasize that experiments for these network state is done with the command line parameters:

- $WINDOW\_SIZE = 30$
- $NUMBER\_OF\_TRIAL\_FOR\_ADDRESS\_TRANSFER = 1$

The main reason underlying this performance increase of our method is related to the fact that it is not a connection-oriented protocol. TCP is an connection-oriented protocol. As a result, it suffers from spending an extra time at the beginning of the transfer. Since our protocol does not have to spend this time, transfer is completed within a smaller time.

The conclusions can also be supported by the Table 1.

Experiments	**TCP**		**UDP**	
	Conf. Interval	**Mean**	**Conf. Interval**	**Mean**
No rules	(0.886, 0.965)	0.926	(0.783, 0.805)	0.794
Packet loss %0	(0.900, 1.125)	1.012	(0.785, 0.807)	0.796
Packet duplication %0	(0.858, 0.956)	0.907	(0.796, 0.837)	0.816
Packet corruption %0	(0.920, 1.016)	0.968	(0.780, 0.805)	0.792

Table 1: 95% Confidence intervals and Means for Nearly Perfect Network State

Network State with Packet Loss

In this subsection, we will analyze the case in which packets are lost with different probabilities and this is the case in which our approach made a huge difference. It performs far better than TCP in networks in which packets are lost and the difference becomes sharper as the probability of packet loss increases.

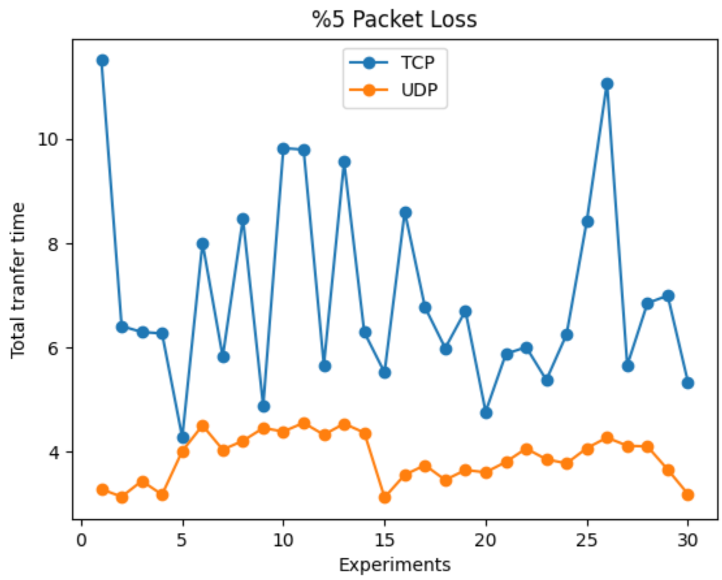

Figure 7: Total Transfer Time for TCP vs UDP - Packet Loss 5%

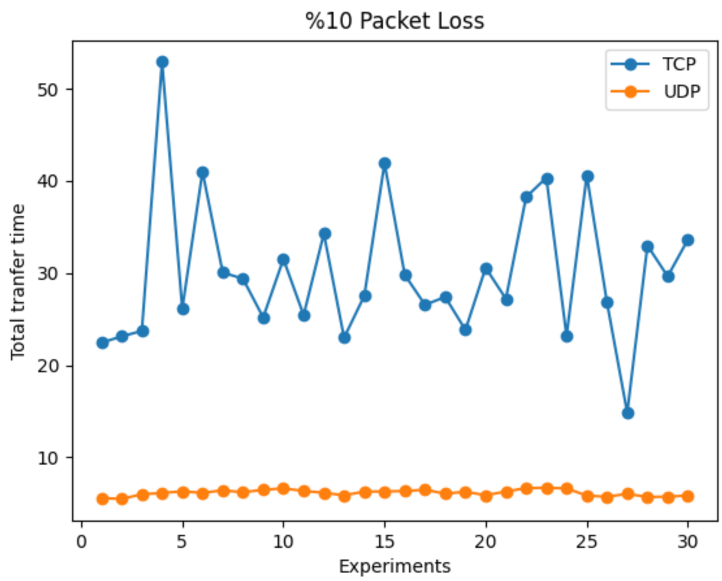

Figure 8: Total Transfer Time for TCP vs UDP - Packet Loss 10%

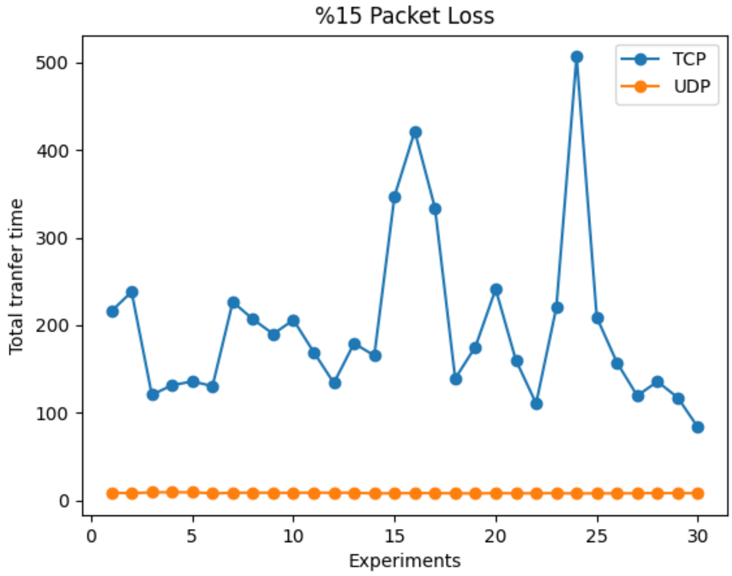

Figure 9: Total Transfer Time for TCP vs UDP - Packet Loss 15%

As it can be seen from the Figure 7, 8 and 9, our object (file) transfer protocol over UDP performs far better (faster) than TCP and this difference of the transmission speed gets larger as the probability of packet loss increase. First of all, we need to emphasize that experiments for these network state is done with the command line parameters:

- $WINDOW\_SIZE = 30$
- $NUMBER\_OF\_TRIAL\_FOR\_ADDRESS\_TRANSFER = 1000$

The main reason underlying this performance increase of our method is related to the fact that the server is likely to send lots of unnecessary copies of the same packet and this reduces the waiting time by the client caused by lost packets and waiting time by the server exposed by the lost acknowledgements. However, this bring some drawbacks despite of such huge performance increase. Since the server is likely to send too much unnecessary copies of the same package this may cause waste of computational power and network congestion. As trade-off is the case in any engineering problem, it is the engineer's, who will use these protocols, choice to sacrifice computation power or transmission speed. Results are also supported in Table 2.

Experiments	**TCP**		**UDP**	
	Conf. Interval	**Mean**	**Conf. Interval**	**Mean**
Packet loss %5	(6.273, 7.678)	6.976	(3.709, 4.047)	3.878
Packet loss %10	(27.227, 32.991)	30.109	(6.040, 6.288)	6.164
Packet loss %15	(162.045, 232.914)	197.480	(8.005, 8.309)	8.157

Table 2: 95% Confidence intervals and Means for Packet Loss Experiments

Network State with Packet Corruption

In this subsection, we will analyze the case in which packets are corrupted with different prob abilities and. It performs better than TCP in networks in which packets are lost and the difference becomes sharper as the probability of packet corruption increases.

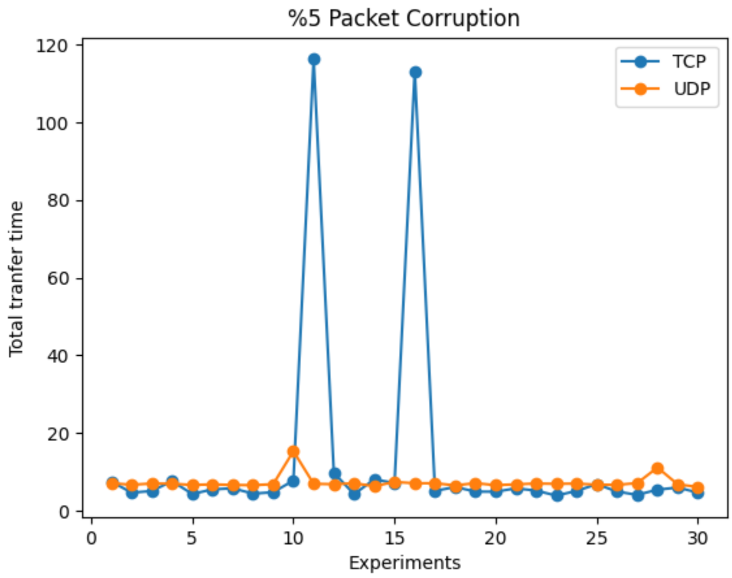

Figure 10: Total Transfer Time for TCP vs UDP - Packet Corruption 5%

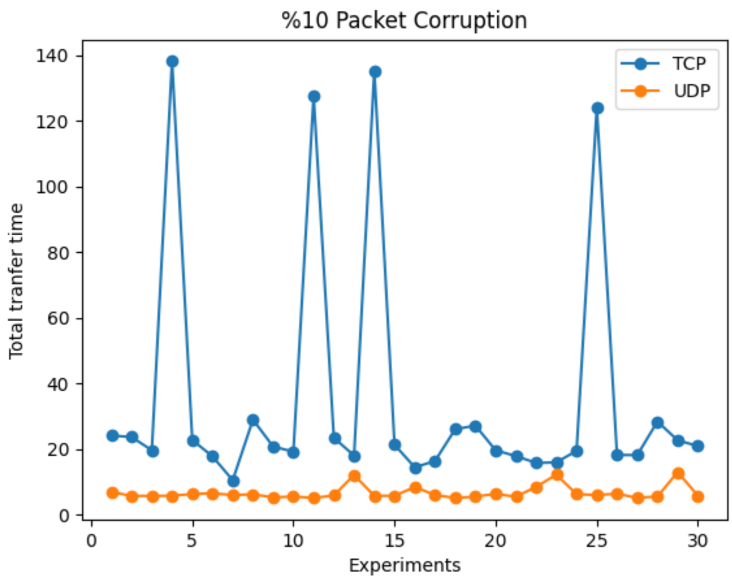

Figure 11: Total Transfer Time for TCP vs UDP - Packet Corruption 10%

We observed that during our experiments we have not received any corrupted packages in both sides of our protocol. Then, we learned that corrupted packages are dropped by the operating system and is not transmitted other side of the communication. Because of this reason, this case is quite similar to Packet Loss case and the performance increase can be explained by the same reasons as the reasons for Packet Loss case.

We need to emphasize that experiments for these network state is done with the command line parameters:

- $WINDOW\_SIZE = 30$
- $NUMBER\_OF\_TRIAL\_FOR\_ADDRESS\_TRANSFER = 1000$

Experiments	**TCP**		**UDP**	
	Conf. Interval	**Mean**	**Conf. Interval**	**Mean**
Packet corruption %5	(2.624, 23.307)	12.965	(6.610, 7.905)	7.258
Packet corruption %10	(20.889, 49.695)	35.292	(5.987, 7.559)	6.773

Table 3: 95% Confidence intervals and Means for Packet Loss Experiments

Network State with Packet Duplication

In this subsection, we will analyze the case in which packets are corrupted with different prob- abilities and our method performs worse than TCP in networks in which packets are lost. However, this defect is negligible in most cases considering the increase in other cases as explained above.

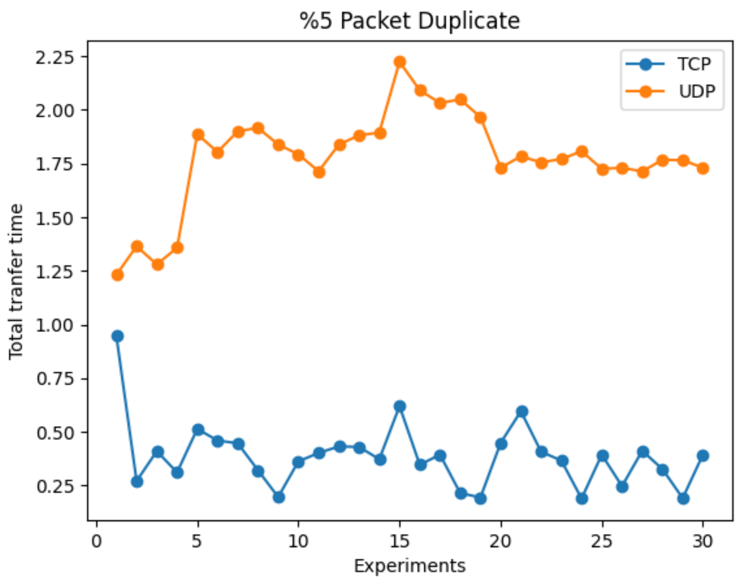

Figure 12: Total Transfer Time for TCP vs UDP - Packet Duplication 5%

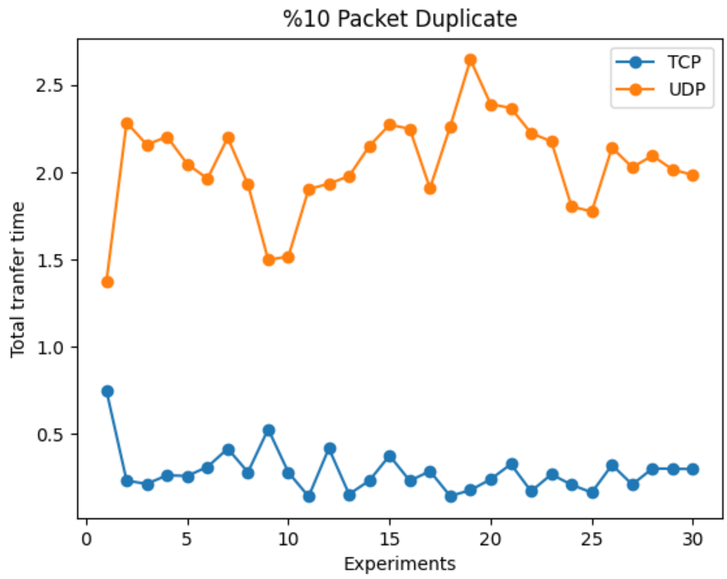

Figure 13: Total Transfer Time for TCP vs UDP - Packet Duplication 10%

TCP handles Packet Duplication in the same way we do. Both of them use ACK and SEQ number to eliminate duplication. However, since we may send too many extra copies to handle packet loss and packet corruption which is not happening in this case, our method's performance gets lower in this case.

Experiments	**TCP**		**UDP**	
	Conf. Interval	**Mean**	**Conf. Interval**	**Mean**
Packet duplication %5	(0.330, 0.444)	0.387	(1.694, 1.862)	1.778
Packet duplication %10	(0.238, 0.331)	0.284	(1.948, 2.151)	2.049

Table 4: 95% Confidence intervals and Means for Packet Loss Experiments

We need to emphasize that experiments for these network state is done with the command line parameters:

- $WINDOW\_SIZE = 30$
- $NUMBER\_OF\_TRIAL\_FOR\_ADDRESS\_TRANSFER = 1000$

Network State with Delay

In this subsection, we will analyze the case in which packets are delayed with different probabilities and our method performs worse than TCP in networks in which packets are delayed. However, this defect is negligible in most cases considering the increase in other cases as explained above.

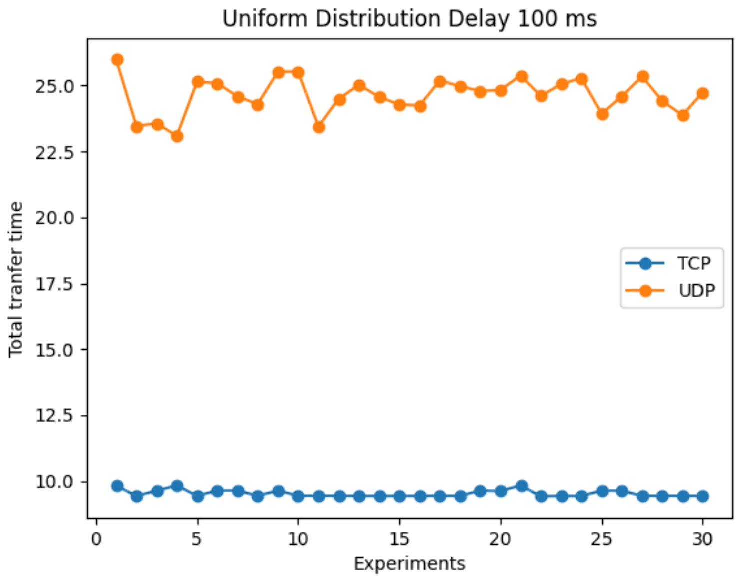

Figure 14: Total Transfer Time for TCP vs UDP - Uniform Distribution Delay 100 ms

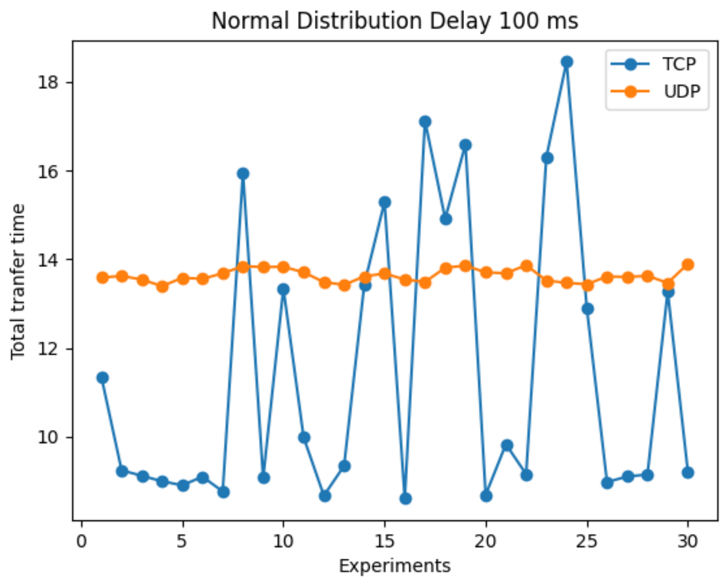

Figure 15: Total Transfer Time for TCP vs UDP - Normal Distribution Delay 100 ms

In this case our method performs noticeably slower than TCP. The main reason behind this fact is that our method (specifically server side) too many times falls into the loop in which it is trying to collect missing acknowledgement numbers (please refer to Algorithm 2). Consequently, it wastes too much time on this process. However, we need you to take your attention to the fact that although delays are coming from normal distribution, our method's total transfer time is as if delays are coming from uniform distribution in Figure 15. The main transfer is again the fact that our method's server sends too much copies of the packets in this case and since it does not wait for each copy for a long time, it is bounded by the timeout we specified in the code. Therefore, it behaves like a uniform distribution. If you need to have transmission time with small small standard deviation, then out method might be a good choice over UDP.

Experiments	**TCP**		**UDP**	
	Conf. Interval	**Mean**	**Conf. Interval**	**Mean**
Uniform distribution	(9.476, 9.577)	9.526	(24.387, 24.907)	24.647
Normal distribution	(10.243, 12.615)	11.429	(13.579, 13.689)	13.634

Table 5: 95% Confidence intervals and Means for Packet Loss Experiments

As you can see in the Table 5, our method performs more stable in Normal distribution since its confidence interval is smaller.

We need to emphasize that experiments for these network state is done with the command line parameters:

- $WINDOW\_SIZE = 210$
- $NUMBER\_OF\_TRIAL\_FOR\_ADDRESS\_TRANSFER = 1000$

Appendix

Exp. No.	**TCP**	**UDP**
1	0.93194	0.72226
2	0.96012	0.78433
3	0.88309	0.80017
4	0.88918	0.80030
5	0.71879	0.81270
6	1.00355	0.80639
7	0.99666	0.77873
8	0.78457	0.79820
9	0.88284	0.79283
10	0.71971	0.77788
11	0.93766	0.77763
12	0.76525	0.78763
13	0.81470	0.79363
14	1.00695	0.78260
15	0.84771	0.79102
16	0.97298	0.80182
17	0.99559	0.78561
18	1.01235	0.83471
19	1.03998	0.78397
20	0.96789	0.77004
21	1.00657	0.76994
22	1.01342	0.84753
23	1.03519	0.76472
24	0.86083	0.76642
25	0.95938	0.79773
26	1.05669	0.77787
27	1.03461	0.80188
28	0.74041	0.82189
29	1.07683	0.78495
30	0.85261	0.90034

Table 6: Experiment Results (No Rules)

Exp. No.	TCP				UDP			
	0%	**5%**	**10%**	**15%**	**0%**	**5%**	**10%**	**15%**
1	2.476	11.514	22.447	215.851	0.731	3.277	5.596	8.501
2	0.975	6.409	23.126	238.017	0.840	3.131	5.496	7.879
3	0.893	6.294	23.721	120.376	0.789	3.434	6.007	8.945
4	1.151	6.268	52.915	131.279	0.795	3.172	6.139	8.999
5	0.980	4.270	26.131	135.665	0.789	4.009	6.327	9.012
6	0.985	8.007	41.005	130.336	0.795	4.502	6.155	7.770
7	1.011	5.822	30.060	226.499	0.772	4.035	6.433	8.338
8	0.753	8.473	29.379	206.728	0.786	4.205	6.221	8.612
9	0.952	4.881	25.184	189.819	0.795	4.451	6.489	8.409
10	0.960	9.827	31.521	205.857	0.818	4.384	6.642	8.394
11	1.083	9.799	25.437	169.279	0.802	4.550	6.364	8.600
12	0.984	5.661	34.279	133.888	0.789	4.322	6.170	8.446
13	0.977	9.564	22.983	179.189	0.803	4.535	5.892	8.557
14	1.031	6.289	27.529	165.216	0.794	4.363	6.293	7.656
15	0.944	5.518	41.908	347.378	0.796	3.118	6.308	7.937
16	0.981	8.608	29.808	421.402	0.778	3.552	6.355	8.086
17	1.032	6.771	26.556	333.829	0.810	3.738	6.496	8.032
18	0.920	5.987	27.405	139.046	0.787	3.454	6.064	7.863
19	0.983	6.703	23.831	175.253	0.800	3.645	6.284	7.601
20	0.972	4.764	30.571	241.100	0.793	3.600	5.893	8.107
21	0.984	5.874	27.211	159.957	0.785	3.800	6.307	7.821
22	1.036	6.005	38.244	110.495	0.783	4.058	6.668	7.868
23	0.933	5.374	40.340	220.096	0.783	3.850	6.714	8.029
24	1.036	6.250	23.190	506.592	0.803	3.772	6.636	7.636
25	0.839	8.424	40.610	208.795	0.775	4.065	5.859	7.838
26	1.097	11.080	26.793	157.104	0.920	4.268	5.733	7.738
27	1.096	5.655	14.840	119.337	0.800	4.111	6.051	7.937
28	1.031	6.850	32.982	135.500	0.793	4.097	5.706	8.137
29	0.718	6.995	29.644	117.243	0.777	3.654	5.751	7.972
30	0.561	5.338	33.629	83.269	0.788	3.184	5.863	7.984

Table 7: Experiment Results (Packet Loss)

Exp. No.	TCP			UDP		
	0%	**5%**	**10%**	**0%**	**5%**	**10%**
1	0.969	0.946	0.750	0.720	1.232	1.372
2	0.919	0.271	0.233	0.783	1.365	2.285
3	1.068	0.410	0.215	0.811	1.280	2.158
4	0.469	0.311	0.264	0.787	1.358	2.204
5	0.943	0.513	0.260	0.828	1.887	2.047
6	0.705	0.460	0.312	0.859	1.804	1.964
7	0.755	0.447	0.416	0.831	1.901	2.198
8	0.891	0.321	0.282	0.808	1.916	1.930
9	0.973	0.196	0.528	0.853	1.839	1.499
10	0.955	0.363	0.280	0.805	1.793	1.518
11	0.965	0.402	0.142	0.738	1.714	1.903
12	1.046	0.432	0.418	0.889	1.839	1.934
13	0.921	0.430	0.155	0.800	1.881	1.978
14	1.048	0.372	0.233	0.921	1.895	2.150
15	0.959	0.620	0.377	0.747	2.225	2.273
16	0.793	0.347	0.232	0.732	2.092	2.249
17	0.970	0.393	0.287	0.890	2.032	1.908
18	0.985	0.215	0.145	0.882	2.049	2.262
19	1.009	0.194	0.179	0.958	1.967	2.645
20	0.839	0.449	0.241	0.805	1.731	2.391
21	0.724	0.596	0.328	0.782	1.783	2.368
22	0.693	0.407	0.172	0.882	1.756	2.224
23	0.861	0.366	0.269	0.846	1.771	2.177
24	1.024	0.189	0.210	0.773	1.807	1.804
25	0.982	0.393	0.165	0.818	1.726	1.775
26	1.034	0.247	0.323	0.804	1.730	2.142
27	0.950	0.412	0.211	0.790	1.714	2.029
28	0.915	0.326	0.303	0.783	1.768	2.095
29	0.890	0.191	0.301	0.789	1.765	2.015
30	0.955	0.394	0.300	0.780	1.731	1.985

Table 8: Experiment Results (Packet Duplication)

Exp. No.	TCP			UDP		
	0%	**5%**	**10%**	**0%**	**5%**	**10%**
1	0.992	7.513	24.215	0.717	7.080	6.990
2	1.048	4.647	23.763	0.722	6.693	5.852
3	0.961	5.089	19.723	0.853	7.006	5.812
4	0.911	7.617	138.204	0.797	6.955	5.887
5	0.965	4.309	22.868	0.808	6.639	6.387
6	0.894	5.505	17.855	0.784	6.789	6.646
7	1.032	5.780	10.609	0.801	6.633	6.071
8	1.077	4.402	29.040	0.800	6.579	6.321
9	1.127	4.762	20.851	0.803	6.835	5.377
10	1.060	7.775	19.295	0.783	15.327	5.628
11	1.072	116.251	127.678	0.779	6.933	5.204
12	1.076	9.739	23.511	0.781	6.832	5.958
13	1.092	4.282	18.117	0.795	7.071	12.230
14	1.043	8.062	135.225	0.782	6.323	5.828
15	1.045	7.146	21.413	0.790	7.412	5.879
16	0.876	113.174	14.525	0.785	7.084	8.525
17	1.003	5.099	16.466	0.775	7.082	6.115
18	0.551	6.116	26.136	0.799	6.529	5.186
19	0.861	4.921	27.232	0.866	7.101	5.619
20	0.981	4.903	19.707	0.786	6.583	6.449
21	0.995	5.761	17.965	0.764	6.801	5.586
22	0.673	5.175	15.884	0.776	7.004	8.567
23	0.723	3.898	16.037	0.770	6.929	12.293
24	0.961	5.104	19.550	0.773	6.988	6.342
25	1.059	6.834	124.058	0.796	6.705	6.102
26	1.013	4.938	18.322	0.820	6.671	6.472
27	0.999	4.080	18.248	0.890	7.113	5.245
28	1.017	5.427	28.493	0.797	11.132	5.654
29	0.912	5.971	22.744	0.795	6.786	13.059
30	1.022	4.670	21.027	0.780	6.124	5.900

Table 9: Experiment Results (Packet Corruption)

Exp. No.	TCP		UDP	
	Uniform Dist.	Normal Dist.	Uniform Dist	Normal Dist.
1	9.827	11.350	25.990	13.590
2	9.431	9.242	23.466	13.626
3	9.632	9.119	23.568	13.543
4	9.831	8.995	23.091	13.396
5	9.433	8.905	25.149	13.583
6	9.640	9.098	25.089	13.560
7	9.638	8.765	24.591	13.686
8	9.437	15.958	24.295	13.845
9	9.635	9.074	25.532	13.828
10	9.438	13.345	25.530	13.833
11	9.442	9.984	23.447	13.707
12	9.435	8.672	24.497	13.488
13	9.434	9.343	25.032	13.424
14	9.431	13.436	24.571	13.616
15	9.431	15.295	24.284	13.680
16	9.434	8.606	24.245	13.547
17	9.436	17.126	25.190	13.490
18	9.434	14.918	24.979	13.811
19	9.634	16.592	24.791	13.859
20	9.622	8.689	24.832	13.707
21	9.829	9.830	25.382	13.686
22	9.420	9.157	24.607	13.867
23	9.432	16.303	25.057	13.524
24	9.430	18.457	25.286	13.470
25	9.636	12.904	23.933	13.438
26	9.631	8.977	24.591	13.614
27	9.437	9.103	25.345	13.604
28	9.436	9.147	24.414	13.627
29	9.435	13.277	23.877	13.469
30	9.433	9.204	24.748	13.900

Table 10: Experiment Results (Delay 100 ms)