CS3543

Better RDT over UDP

Computer Networks -2

Team Details

We are a team of 5 people.

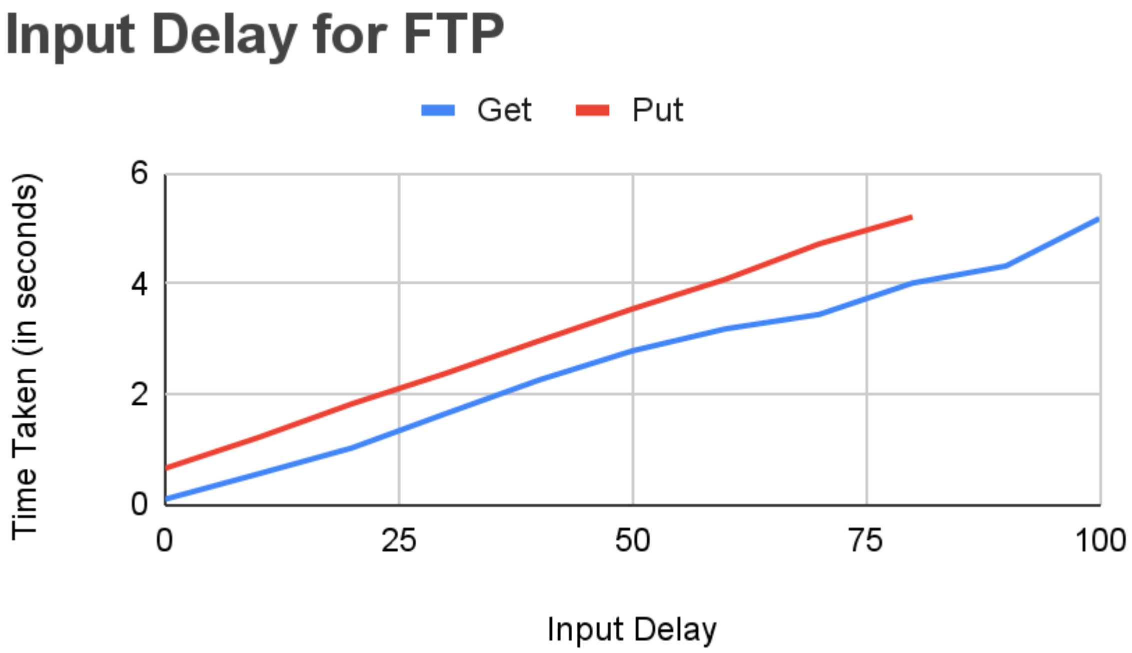
Bhavanam Sujeeth Kumar Reddy	-	ES19BTECH11022
Mukkavalli Bharat Chandra	-	ES19BTECH11016
Grandhi Sai Sidhardha	-	CS19BTECH11050
Mylavarapu Sri Akhil	-	ES19BTECH11014
Krishn Vishwas Kher	-	ES19BTECH11015

Task: 1 Preparation and Preliminary Study

The current FTP we are using uses TCP which is very sensitive to delay and packet loss. Let us observe how the time taken for FTP changes with multiple conditions of Packet Loss and Input Delay.

Input Delay for FTP

We have first tested FTP speeds by varying delay by 10ms from 0 to 100ms without any speed cap.



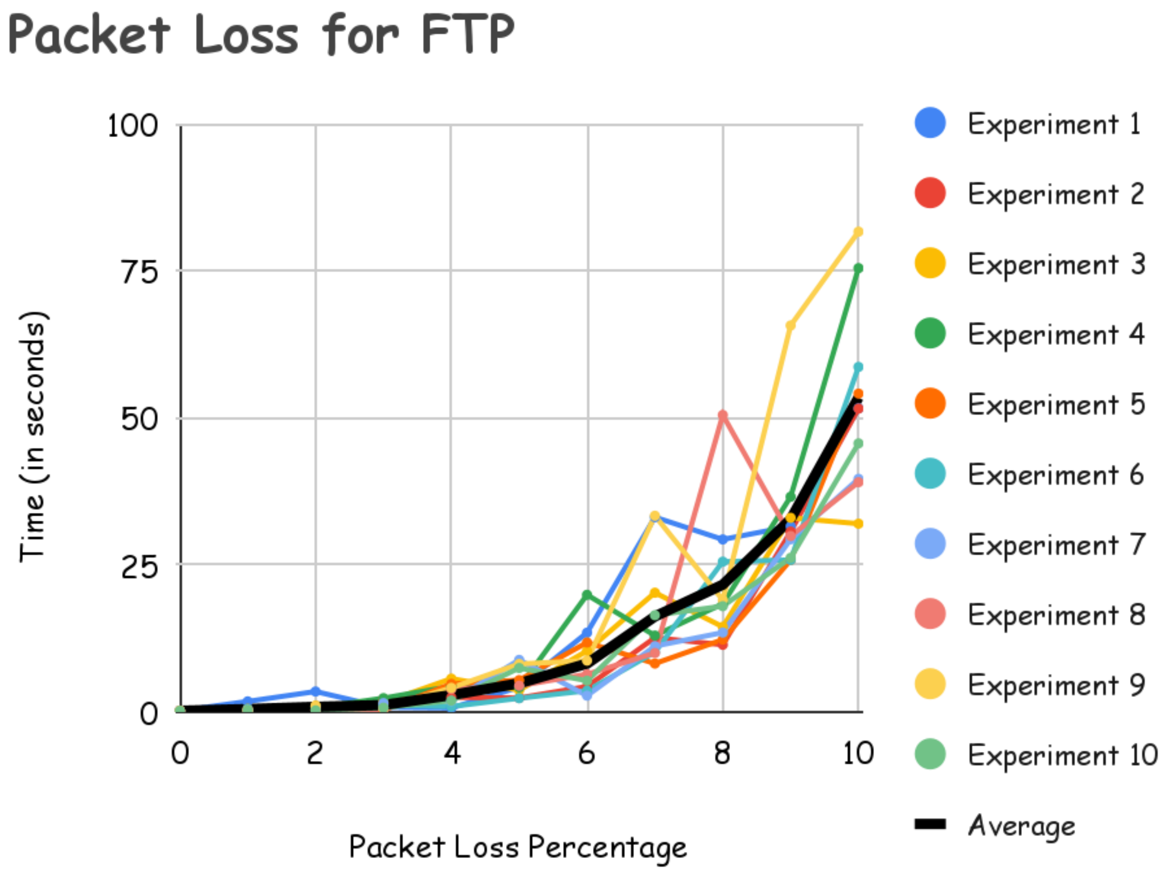
From the results, we can see that the time taken for file transfer increases linearly with a linear increase of delay as the packets only take more time to transfer and no packets are retransmitted.

We also notice that for a fixed delay the time taken does not vary much for every experiment which is in line with our theory as no packets are retransmitted and each packet takes a similar time to be sent in every experiment with a fixed delay.

Upload speeds are generally faster than download speeds and that is also seen in our observations, get <file> takes slightly lesser time when compared to put <file>.

Packet Loss for FTP

Now we test for packet loss by varying loss by 1% from 0 to 100% without any speed cap. As the time taken keeps varying a lot with each experiment having a fixed loss percent, we repeat the experiment 10 times.



(For the detailed graph and comparisons, visit the linked Google Sheets for better viewing)

Most noticeably at a packet loss percentage of 10%, we can see that the time taken varies a lot with each experiment. As the packets dropped are random, these packets have to be retransmitted, the packets that are retransmitted and the time taken to retransmit them are different with every experiment.

We can see from the graph that even though the time taken varies a lot with each experiment the general trend as loss percent increases is exponential, as more packets are dropped the time taken to retransmit keeps increasing which causes an exponential increase in the time taken.

We tried to fit the average line into a standard curve, and it appears that the time in seconds is exponential in terms of the packet loss (another curve that seemed to fit well is the curve time = c*(packet_loss)^{6}, where c is some constant). This is justifiable especially in the context of TCP, since as packet loss starts happening, according to the TCP protocol, the flow and congestion control protocols also try to adjust the windows accordingly.

50ms Delay and 5% Packet Loss

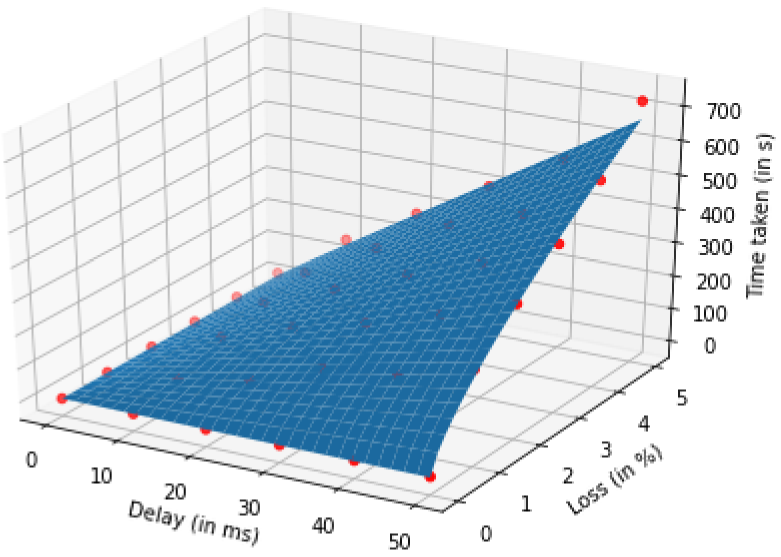
In the below table we can see the data of multiple file transfer protocol runs when limited to a maximum speed of 100Mbits/s and their respective time taken to send the file and the appropriate throughput.

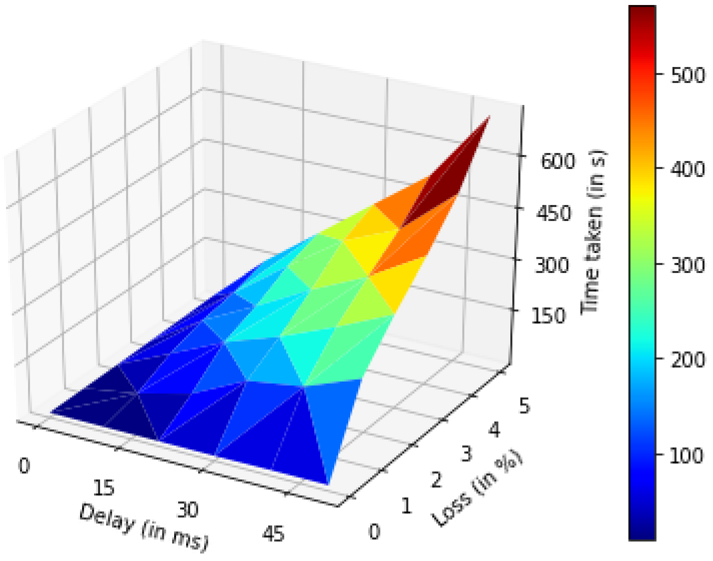
5% Loss and 50ms Delay		
Experiment No.	**Time Taken**	**Throughput (kB/s)**
1	1329.41	77.0266
2	1354.33	75.6093
3	1366.64	74.9282
4	1356.93	75.4644
5	1378.57	74.2798
6	1402.02	73.0374
7	1323.51	77.3702
8	1376.54	74.3894
9	1394.59	73.4265
10	1333.41	76.7955

From the data, we can observe that due to the speed limit, the time taken is significantly higher and Throughput is comparatively low as the loss and delay are high. This is not a viable method of transfer for big files and networks with higher loss and delay.

3D plot of Loss v Delay v Time

We also tested varying packet loss and delay together without any speed cap.





The results are shown in the 3D graph above where the X-axis is delay (in ms), Y-axis is loss and Z-axis is the time taken(in s). We can see that even with a small delay and loss, the time taken is significantly higher than having only delay or only loss.

The 3-D diagram helps us understand the concavity of the surface when both the delay and packet loss are considered. We see that as both the parameters are increased, the time increases more and more and as such even the gradient becomes steeper. Due to fewer points, there are sharp vertices in the surface (and incidentally gives the interesting design), but with more data points, we expect that these vertices get smoothened.

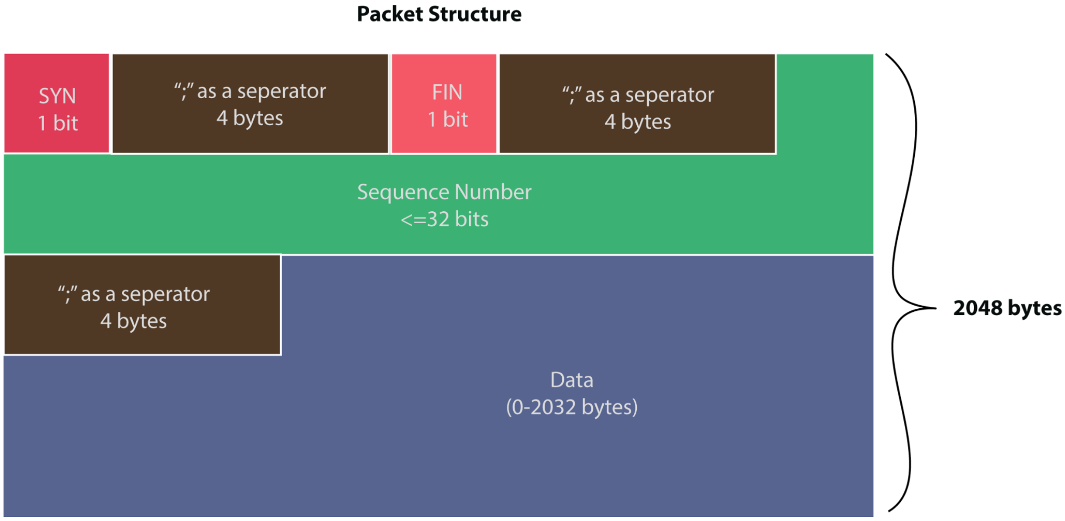
Note: We tested using 3 host VMs and there were no significant differences between experiments on different hosts. We think that this could be because we are ultimately running these experiments on Ubuntu servers and the effect these experiments pose on the host machine seems to be uniform across different host machines.

Task: 2 Implementing "our-UDP-FTP"

Theoretical Aspects:

We have attempted to implement 2 protocols, namely SR and the TCP protocol. We attempted to implement the TCP protocol without the security feature which TCP provides and without flow control (but we do include congestion control).

The most part of these algorithms was implemented without any changes, except for a few of them, which will be clear by our descriptions. We begin by explaining what packet structure we assume, as follows:



We also mentioned that we refrained from implementing checksum finally because:

1. It was taking longer time to transfer the file with it than without it;
2. Was not necessary, since the underlying UDP service would anyway do it.

The idea behind implementing SR was that seemingly it felt to be more efficient than GBN, in the sense that unnecessary retransmissions are avoided in the former as compared to the latter, by buffering out-of-order, but correctly received packets.
The idea behind implementing TCP was since we were not particularly implementing the security features, we thought that it might lead to faster runtimes when transferring files.

Initially, we thought of also using some ML based algorithm in the congestion control part of TCP. The reason behind this was to try to predict a delay or loss even before it happens and adjust the congestion window accordingly so as to prevent it from happening, unlike in normal TCP.
We went through some research papers for this (R1, R2, R3), but many of them seemed to indicate some sort of reinforcement learning based approach, with which we were not particularly conversant with and they also seemed to indicate that RL based approaches don't necessarily improve the runtime but improve other factors.

We eventually thought of using some sort of a gradient boosting based regressor (like XGBoost or LGBM) since usually they are not only highly accurate, but are also very fast at predicting an output. But we didn't want to burden the classifiers with too much data and hence we were planning of receiving data of (cwnd, sampleRTT) for some batch size, and then training over it and using it to predict the optimal value of cwnd when provided with an estimatedRTT.

However, computing the exact loss function still seemed to be a bit tedious and we weren't sure how much it would really help in giving a speedup. We consider a NaiveBayesClassifier also, since we are usually only concerned about the integer part of (cwnd/MSS), but resources seemed to indicate that NaiveBayesClassifier based approaches in this context could possibly not help a lot. We were also out of time to implement so we proceeded to implement the normal congestion control algorithms which TCP usually implements (an AIMD-style approach but with more careful increase when cwnd ≥ ssthresh to avoid congestion).

We also noted that while we need threads for implementing the TCP algorithm (at least as presented in the textbook), we need to be careful while using them as using them incorrectly can lead to increased packet losses/delays, which is against our objective.

The first optimization we do as compared to SR is that we use only 1 thread to keep track of the time, as opposed to our implementation for SR, where we maintain a thread for each packet. We try to perform this optimization for TCP because creating threads poses a significant overhead in itself.

Another issue we need to handle carefully for TCP is that although multiple threads should be executing certain events in parallel, there are certain parts of the events that we felt should be executed sequentially, for which we used a priority queue with timestamps for the events (note that a priority queue also offers a highly efficient time complexity of O(log n)).

We also keep updating the RTO (time out interval) and congestion window in accordance with the dynamically updated estimated RTT values from the client to the server (and back to the client).

We also attempted a few optimizations while implementing SR. Note that these optimizations aren't necessarily very significant speedups as far as the algorithm is concerned, but in practice they helped improve the running time by some significant amount.

These mainly included things such as using inbuilt Python functions for looping or accessing data chunks at a time rather than relying on our own for loops to do the same (ex: using the 'join()' function in Python for writing to the buffer than using a for loop to do it).
We have used Python's dictionaries for buffering as on average, they are highly efficient (O(1) average time complexity).

In addition we would like to mention that the code design was intended to be kept quite modular as it would help in detecting any bugs in the code and help in re-using functions across different protocols.

Testing:

Testing "our-UDP-FTP" without any loss or delay we get these speeds:

On average, we got around 16 seconds for the file transfer of a 100MB file, when we test on the local machine (not the VMs), when using the SR protocol.

These speeds are almost the same as FTP, this is expected as both are performing in ideal situations.

Testing "our-UDP-FTP" with 50ms delay and 5% packet loss:

We haven't experimented with this.

Conclusions:

We see that “our-UDP-FTP” performs roughly on the order of how TCP works under normal conditions. This is because of considerable effort taken over the unreliable and light service of UDP in the implementation of our protocol(s) in order to maintain reliability.