Assignment 3

Reliable Data Transfer with congestion control

Piyush Chauhan 2021CS11010
Shankh Gupta 2021CS50604

A COL334 Homework Assignment

भारतीय प्रौद्योगिकी संस्थान दिल्ली
Indian Institute of Technology Delhi

November 1, 2023

Milestone I

a. Protocol :

We have implemented the following protocol to for reliable data retrieval:

- We have set up an array **DataStream** which maintains all the packets in order. We iterate through the array and send packet request to the server sequentially.
- If we get the request within the timeout period, the packet is processed and stored else we drop the packet and move forward.
- After the completing one iteration, we again iterate to retrieve packets that were dropped in the previous iteration.
- We continue iterating the DataStream until all the packets have been received.

b. Ensuring Reliability :

To ensure that the data is received in a reliable manner, we have set up various **error handling protocols**. These include insuring that the length of the packets received is correct, the offset is same as that was requested by the client, checking for corrupt packets, checking for squishing, etc. Whenever some error is encountered the packet is dropped and can be retrieved in a later iteration.

c. Graphs :

We plotted the following graphs to analyze our protocol. Here all the graphs have Time (in seconds) in the x-axis and Offset in the y-axis. **Requests are shown in blue and Replies are marked in orange**. The graphs plot offset received vs the Time(from starting of connection to the server) at which they were received.

d. Plans for Next Milestone:

Clearly, our current algorithm is a very naive way of getting the data from the server. It receives data at a constant rate, and doesn't take advantage of server's high bandwidth (if available) or adapts to a lower bandwidth. Also it sends only 1 request at a time.
For our next Milestone, we plan to incorporate a **congestion window protocol**, which sends requests in burst and adapts to the server rate to get to an optimal burst size and request rate.

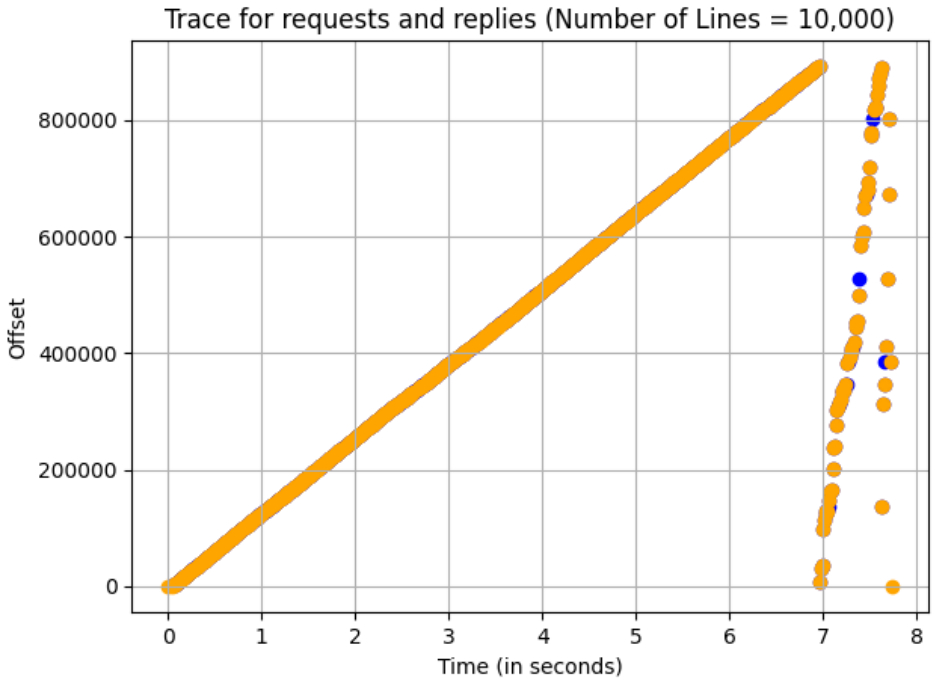

Figure 1: Sequence Number Trace for 10,000 Lines

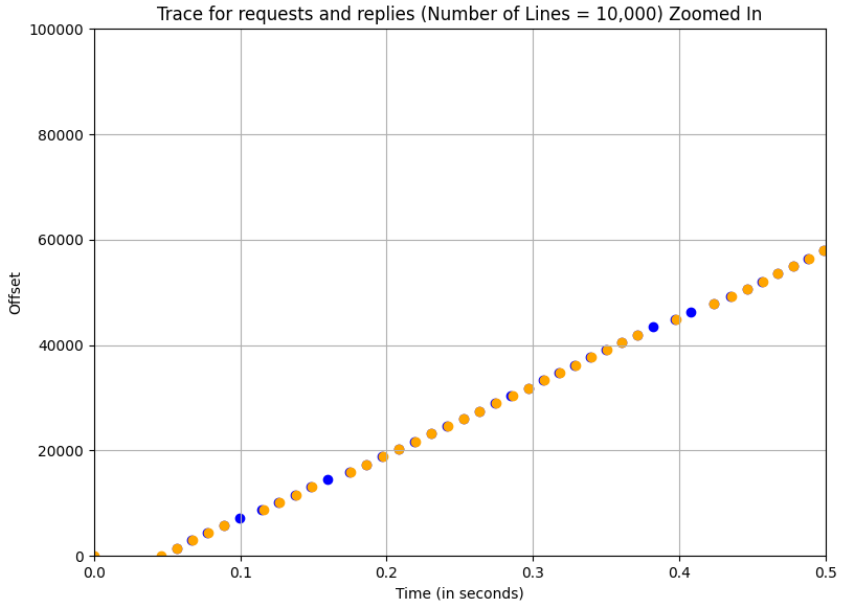

Figure 2: Zoomed-In Sequence Number Trace for 10,000 Lines (upto 500ms)

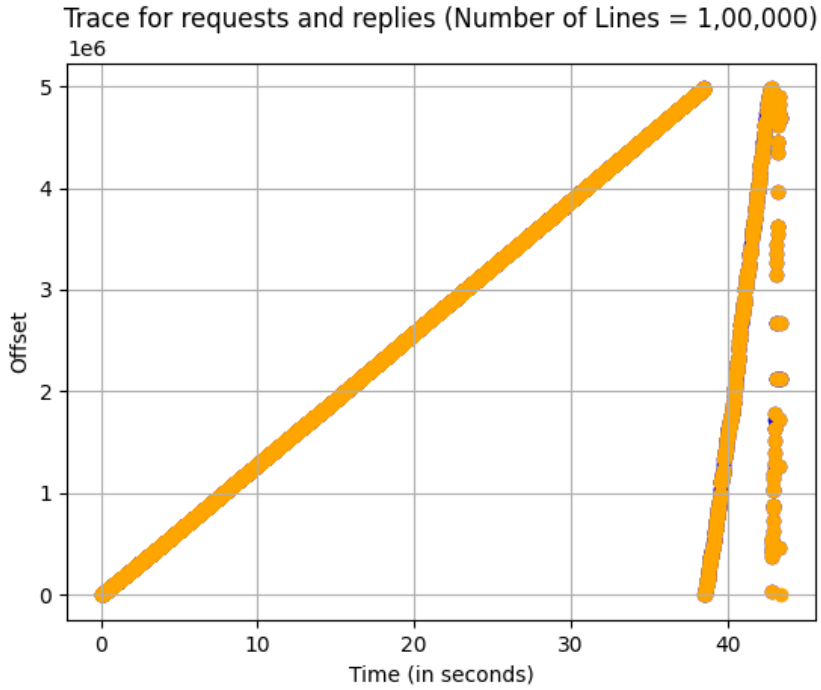

Figure 3: Sequence Number Trace for 1,00,000 Lines

Milestone II

a. Protocol:

The protocol used is very similar to **Selective repeat** version of RDT. Overview is to maintain a congestion window of some fixed size and request the server of these packets simultaneously. Then, if we receive a packet from the server, we mark that as received and send Send-Request of next unsent packet. This goes on unless all the packets are received. Details are as follows :

- We maintain a **congestion window** of constant size 6 and packet size 1448. Initialize by requesting 6 packets of data from server starting from offset 0. Maintain two pointers *start* and *end* do denote start and end points of congestion window.
- (If) The server replies with some packet. We calculate the packet number using the offset and store it at its respective index.
- If the packet number is equal to *start* packet of congestion window then we send a request for packet number $end + 1$. Thereby, increasing window by 1.
- Otherwise, we wait for some **cool-down period** before re-requesting the *start* packet.
- Everytime we receive a packet, we update our RTT according to $equation1$.
- If we get squished, then we wait for 500ms for tokens to regenerate and then send the request for new packets.

b. Estimating the RTT

We are estimating the RTT on fly using EWMA (exponential weighted moving average) keeping $\alpha = 0.825$.

$$estimatedRTT = 0.825 * estimatedRTT + 0.125 * sampleRTT \tag{1}$$

We tried several times getting data from vayu.iitd.ac.in and found estimated RTT to be around 4-5ms. Total time to receive files was 12-15 sec. Graphs are attached in the graphs section.
We also tried the hack-solution suggested in assingment of fixing the RTT. We fixed the RTT to 4ms and ran the code. It took on an average 11 - 12 sec to receive and successfully submit to vayu.iitd.ac.in.
We also tried various sizes of congestion window and found 6 to the one with least penalty.

c. Statistics:

The following Outputs are some sample testcases to show the performance of our protocol.

I. **Requests sent at adaptive Rate (with EWMA protocol)** (tested on server : 10.17.7.134 on Oct 24)

- Result: true
  Time: 12495
  Penalty: 2

- Result: true
  Time: 12759
  Penalty: 2
- Result: true
  Time: 12527
  Penalty: 1
- Result: true
  Time: 12389
  Penalty: 6
- Result: true
  Time: 12566
  Penalty: 3

II. **Requests sent at a Constant Rate aka Hacky Solution (without EWMA protocol)** (tested on server : 10.17.7.134 on Oct 24)

- Result: true
  Time: 10949
  Penalty: 2
- Result: true
  Time: 11249
  Penalty: 14
- Result: true
  Time: 11226
  Penalty: 1
- Result: true
  Time: 10785
  Penalty: 9
- Result: true
  Time: 11401
  Penalty: 5

NOTE: The hacky solution in our testing conditions outperforms the adaptive rate protocol. But this might not always be the case, especially when there is congestion in the network because the constant rate server will keep sending requests at the same rate despite the network congestion whereas the adpative rate protocl will adjust the rate according to the Sample RTT it calculates.

III. **Testcases ran on localhost server** (tested on 100k Lines)

- Result: true
  Time: 18567
  Penalty: 1
- Result: true
  Time: 19455
  Penalty: 1

- Result: true
  Time: 18510
  Penalty: 1

NOTE: with constant rate we were able to achieve time of 16.657 seconds with Penalty : 1.

IV. **Testcases ran on localhost server** (tested on 10k Lines)

- Result: true
  Time: 3235
  Penalty: 3
- Result: true
  Time: 3240
  Penalty: 1
- Result: true
  Time: 3316
  Penalty: 1

NOTE: with constant rate we were able to achieve time of 2.914 seconds with Penalty : 1.

d. Graphs:

We plotted the following graphs to analyze the performance and working of our protocol:

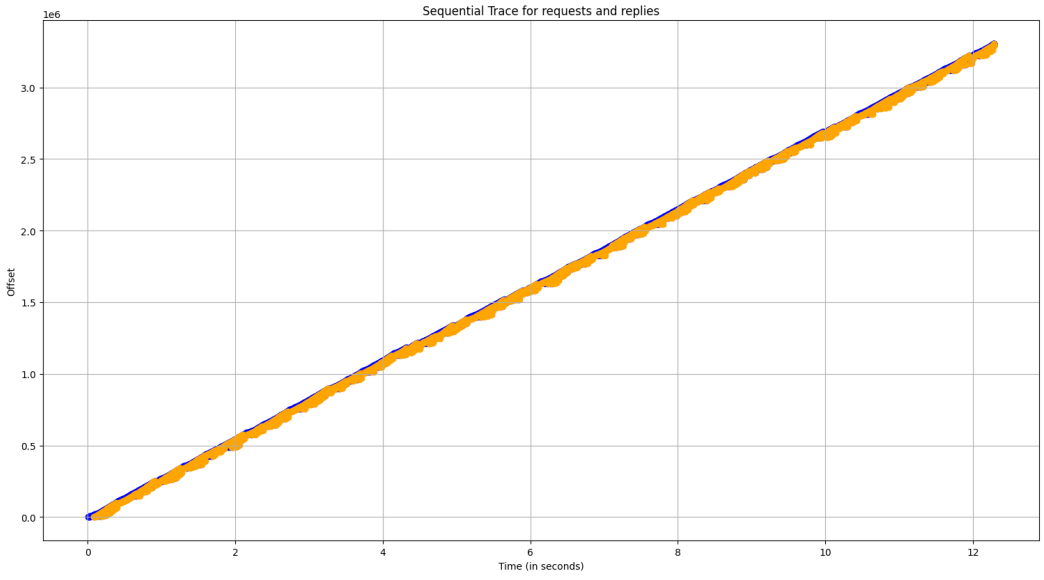

Figure 4: Sequence number trace for downloading the file from Vayu server.

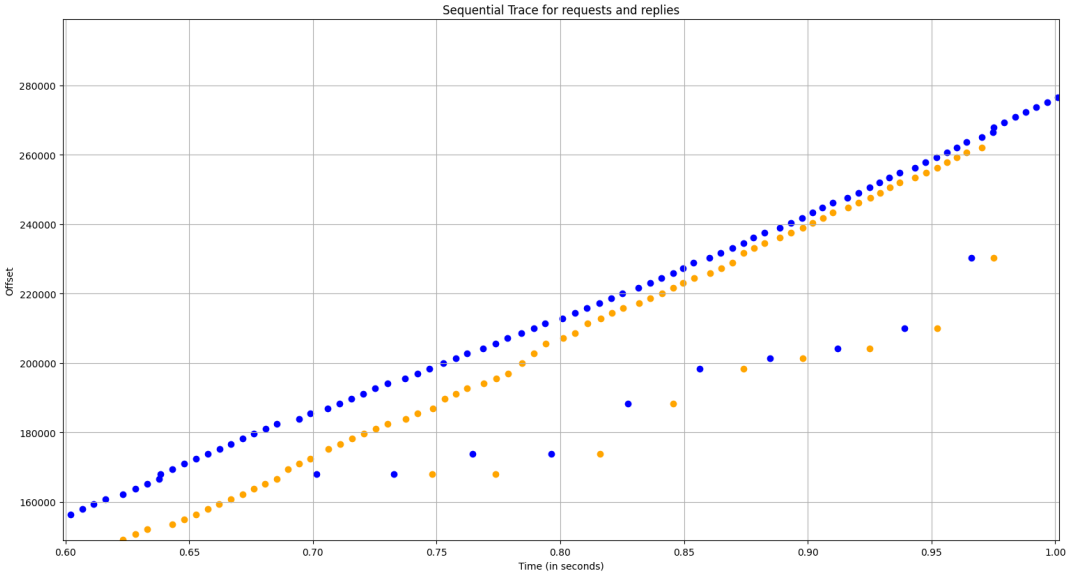

Figure 5: Sequence number trace for downloading file from Vayu server (Zoomed in graph).

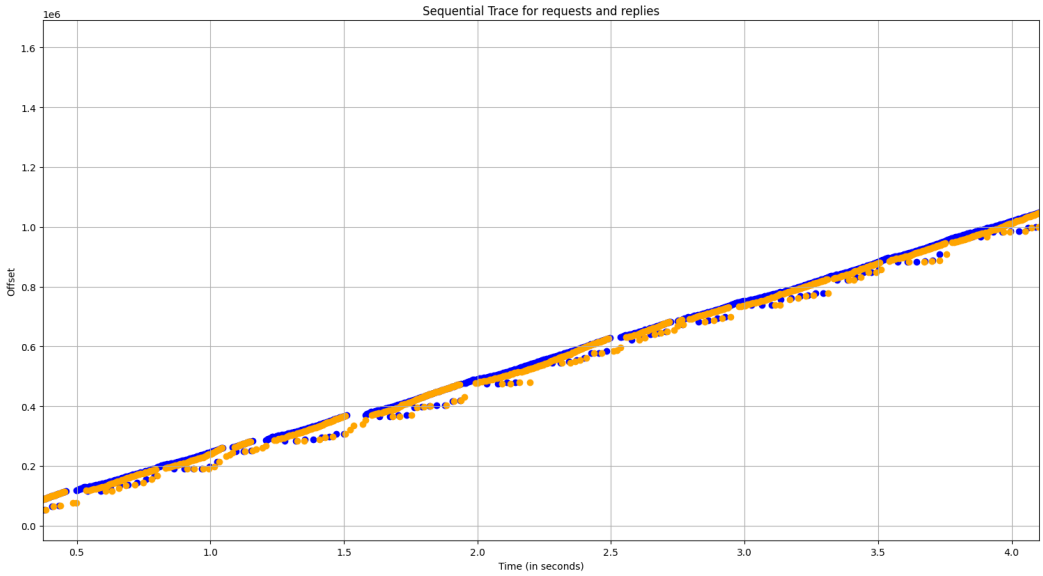

Figure 6: Sequence number trace for downloading files from Localhost server (100K lines)

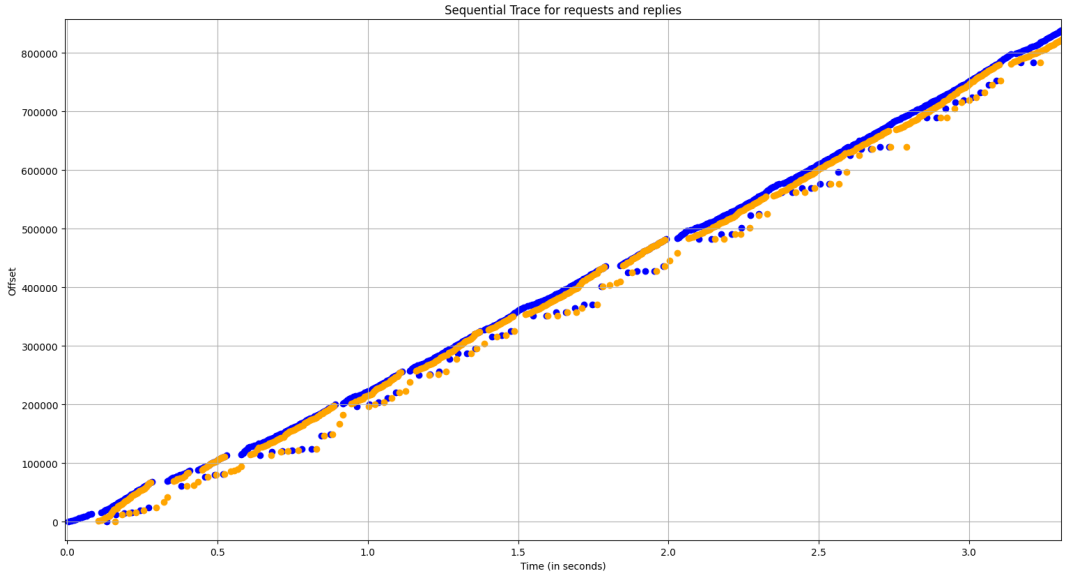

Figure 7: Sequence number trace for downloading files from Localhost server (10K lines)

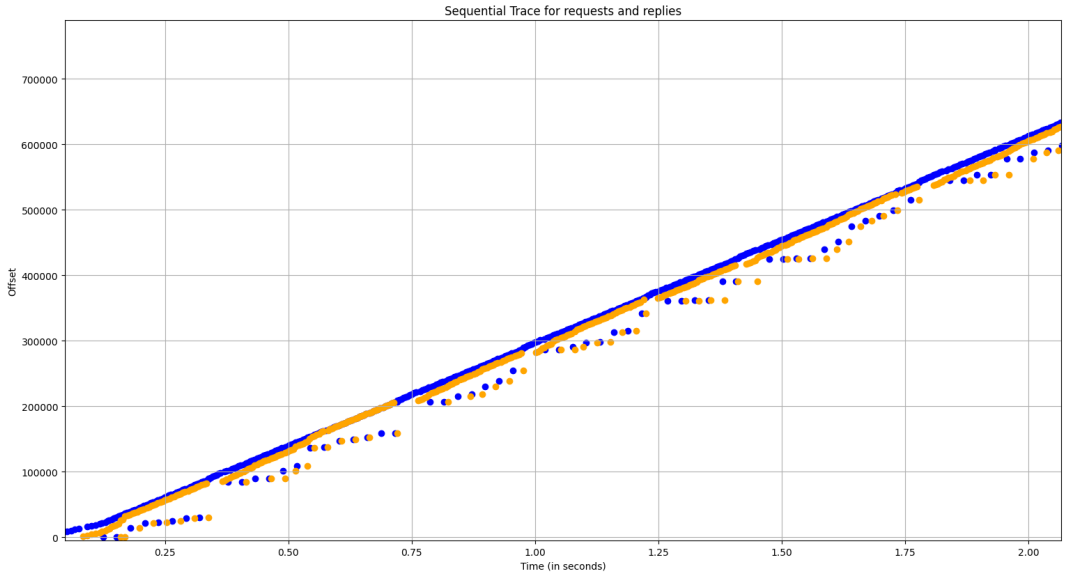

Figure 8: Sequence number trace for downloading files from Vayu server at constant rate without EWMA protocol for adaptive Client-Rate.

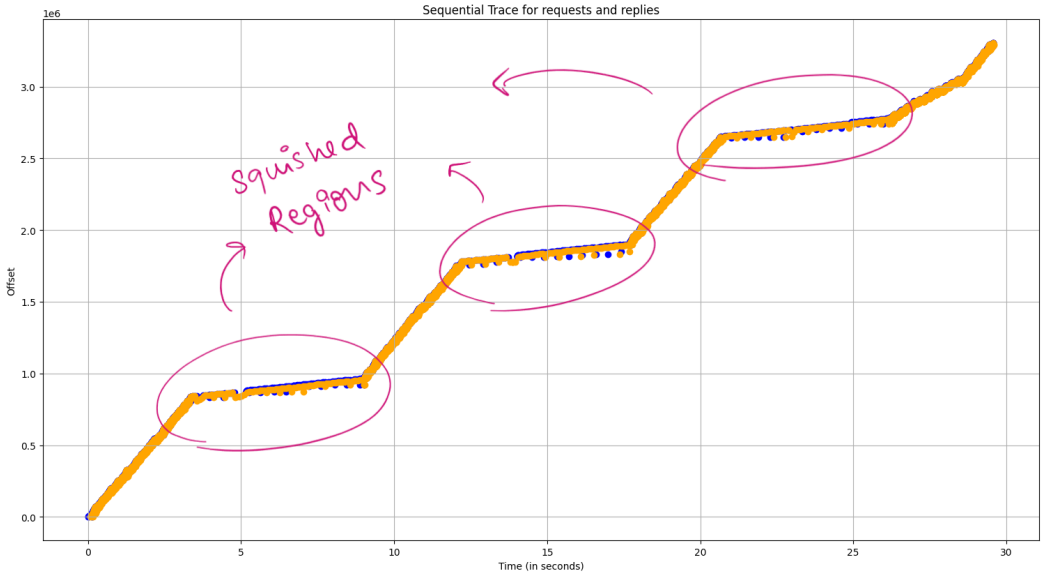

Figure 9: Sequence number trace for downloading files from Vayu server at a faster rate and getting squished.

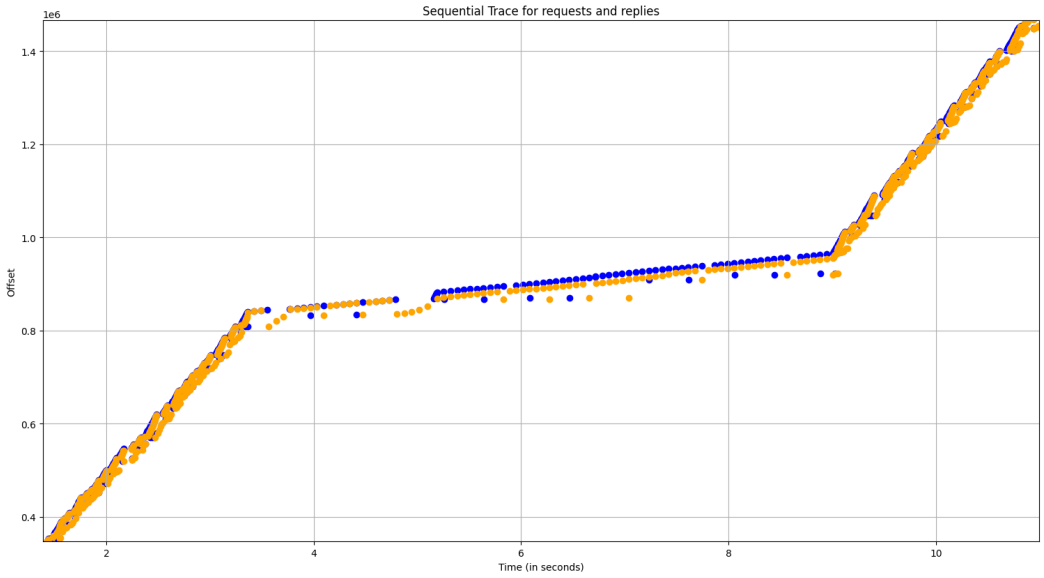

Figure 10: Sequence number trace for downloading files from Vayu server at a faster rate and getting squished (Zoomed in graph).

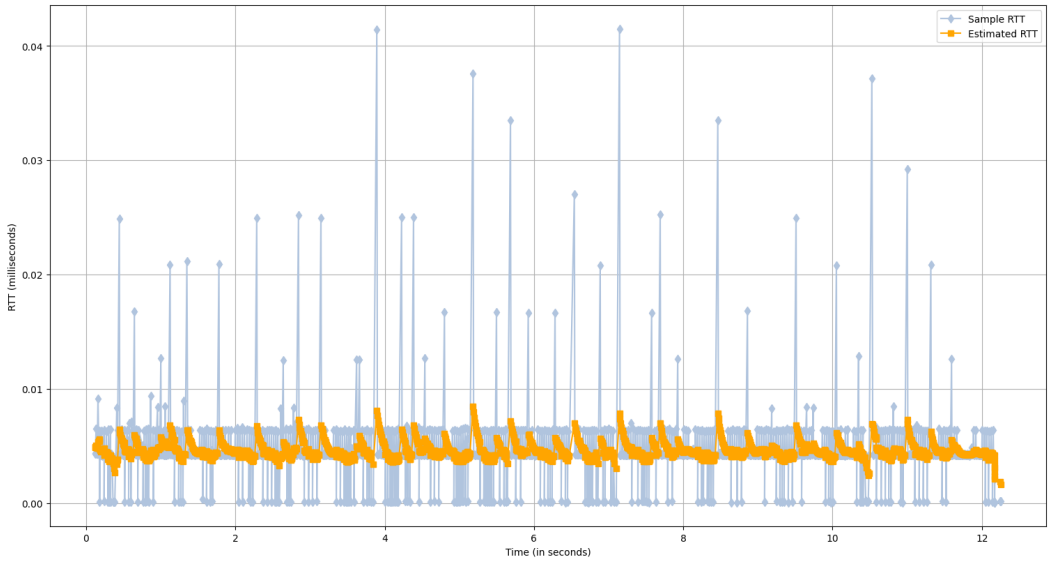

Figure 11: RTT vs Time graph for Estimated and Sample RTT for EWMA RTT Estimation protocol.

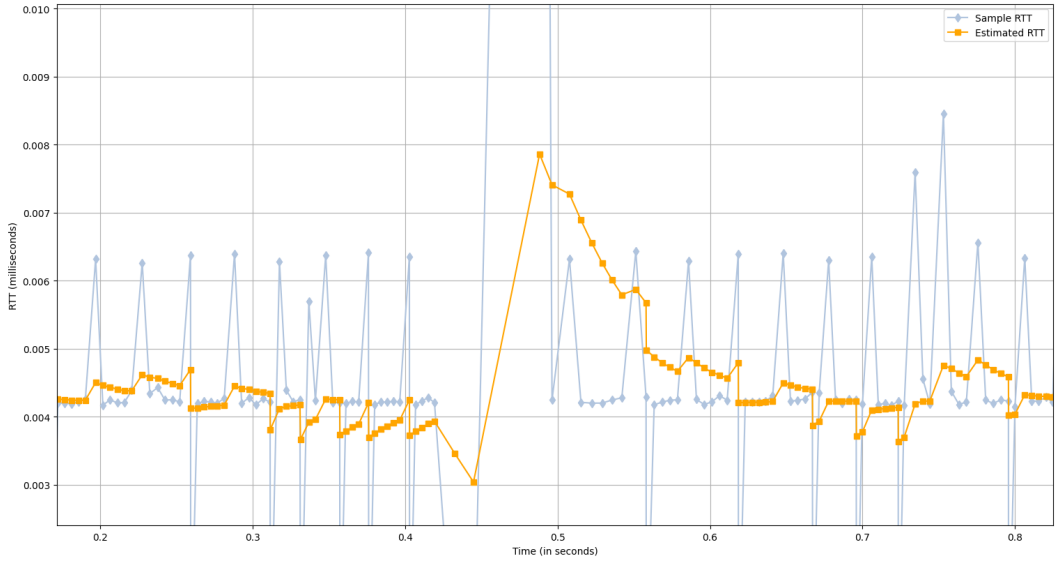

Figure 12: RTT vs Time graph for Estimated and Sample RTT for EWMA RTT Estimation protocol (Zoomed in graph).

d. Comments on the above Traces:

1. `Figure 4.` and `Figure 5.` Shows the graphs for the files received from the Vayu server. The blue dots represent the Send-Request and Orange dots represent the Replies received. As is evident from the graph,

the protocol sends requests one by one and the corresponding replies for that offset is received much later after whole of our window is traversed.

2. Occasionally, there are some send requests sent for a lower offset compared to others (isolated blue dots in `Figure 5.`). These are the requests for the start packet (which was skipped by the server in the first time) sent after the cooldown period.

3. `Figure 6.` and `Figure 7.` are traced while receiving the files (of size 100k lines and 10k lines respectively) from the localhost server. These are similar in nature to the previous plots with the only differnce being that the localhost server is faster in terms of RTT.

4. `Figure 8.` which shows receiving data at a constant rate is also similar to the previous graphs, only it has now a constant slope everywhere because it is sending requests at a fixed rate.

5. `Figure 9.` and `Figure 10.` represent the case when we send requests at a much faster rate and the client experiences squishing. The time period when the server was squished is evident with a significantly lower rate of sending and receiving replies (Marked in `Figure 9.` as well). In the zoomed in graph (`Figure 10.`) you can also see that the graph is more sparse as fewer requests are being serviced by the server.

6. Finally `Figure 11.` and `Figure 12.` are the graphs plotted for the Estimated RTT and the sample RTT throughout the duration of receiving the file from the server. As can be seen in the graph, there are occasional spikes in the sample RTT line chart because of the server's randomly skipping the requests (leading to a temporary rise in Estimated RTT as well). Overall the estimated RTT maintains a nearly stable value with a range withing which it fluctuates so as to adapt to the varying rate at which the server replies because of network congestion.

e. Configurable Parameters in our code:

- **Congestion Window Size:** This dictates the size of our congestion window. For this milestone, we kept it to be constant. We found a window size of 6-7 gives best results and lowest penalties in the long run.

- **Cool-Down period:** This specifies the number of packet replies we wait before sending the request for unacknowledged Start-Packet again. We traverse the window for the cool-down number of packets and keep storing the packets we received at the correct index according to their offset. If we don't find the start-packet upto this point, we reissue a request for it. This approch prevents sending duplicate requests for a packet that was received in the buffer but was shuffled/reordered. We kept the cooldown period to be near half of the window size.

- **Timeout Period:** Timeout period defines the time our client waits for a reply before getting a socket timeout error. This was set roughly equal to the RTT.

- **Freeze Time:** Freeze time is the amount of time our client waits after sending a request to allow the server to regenerate tokens and not overwhelm the server with rapid requests. We keep updating the freeze time in our protocol depending on the RTT using the EWMA equation (*equation*1).

Milestone III

a. Protocol

In the final milestone, we have deployed the protocol which sends requests in bursts of variable sizes, which are learned during the process according the reply rate of server, over multiple iterations untill all the packets are received. The details of the protocol are as follows :

- Send requests in a burst and store all the replies received ignoring the ones which were lost or got corrupted during the transmission.
- Keep sending requests in a burst for all the packets until every packet has been iterated.
- Start iterating again but this time over the packet indices which couldn't be received in the previous iteration.
- The burst size is updated after each burst depending on the reply rate of the server. This is done by checking whether any packet requested in the previous burst was dropped (or corrupted) or not. If that's the case, then we reduce the burst size by half else we increase the burst size by 1. In this way we prevent the algorithm from being squished.
- In addition to varying burst sizes we are also learning the RTT on the fly using EWMA (Exponential Weighted Moving Average) equation in a similar manner as was done in Milestone 2. A wait time of some multiple of RTT is kept between sending and receiving processes so as to allow the server to regenerate tokens. The following equation was used to sample the RTT on the fly:

$$estimatedRTT = 0.825 * estimatedRTT + 0.125 * sampleRTT \tag{2}$$

b. Why Change the Protocol in Milestone 3?

In milestone 3, we have adopted a different protocol than what we did in milestone 2. The following are the reasons for the change :

- The protocol we adopted above allowed us more flexibility to adapt to a variable rate compared to the previous protocol which was more difficult to train as it was sending requests in a sequential manner, the rate of which were difficult to learn and adapt according to a variable server.
- The milestone 2 protocol works very well in a constant rate server and outperforms the burst sending protocol (with constant burst size). The reason for this is because it keeps sending requests continuously, as soon as a packet is received, and not wait for all of the burst.
- Additionally the previous protocol required very less timeouts and waiting time between multiple requests as it was sending requests one by one only, and it was easy to adapt to the RTT using our EWMA protocol.
- But these things do not hold for variable rate server because now the token generation rate is varying, and the previous protocol could only adapt to varying RTT and it was difficult to adapt to varying rate by server. So we were getting our best results with our current protocol only, which is sending requests in a burst. Here if the server rate drops, we can slow down and adapt the burst size to avoid squishing.
- For comparison, Using our previous protocol, the best we were able to achieve was 25-30 seconds of time to receive entire file with a high penalty of 60-90.
  Our current protocol gets the whole file in 20-22 seconds with a penalty of less than 40-50.

c. Congestion Window vs Burst sizes

As discussed earlier, we switched our protocol from maintaining a selective repeat congestion window protocol to sending out requests in bursts. We tried to improve our previous protocol by learning the rate of sending data using AIMD approach. We were incrementing additively the rate and decreasing the rate by half whenever we encountered multiple requests skipped continuously(Hinting that the bucket might be empty). We present here some comparison in running time of the two:

Burst size protocol	
Time (in seconds)	Penalties
28.489	81
25.944	78
27.522	64

Congestion Window Selective Repeat	
Time (in seconds)	Penalties
21.731	42
20.147	48
20.477	31

The reason of the above observations is as explained in above part (b.).

d. Running Times :

The following Outputs are some sample testcases to show the performance of our protocol.

I. **Running Time with the Vayu server)** (tested on server : 10.17.6.5 on Nov 1st)

- Result: true
  Time: 21.731
  Penalty: 42
- Result: true
  Time: 20.147
  Penalty: 48
- Result: true
  Time: 20.477
  Penalty: 31
- Result: true
  Time: 22.521
  Penalty: 29
- Result: true
  Time: 21.032
  Penalty: 47

II. **Requests but learning with higher penalties** (tested on server : 10.17.6.5 on Nov 1st)

- Result: true
  Time: 16.502
  Penalty: 68

- Result: true
  Time: 19.922
  Penalty: 79
- Result: true
  Time: 17.786
  Penalty: 87
- Result: true
  Time: 18.901
  Penalty: 81

NOTE: These are the best values we could achieve if we allow higher penalties in our results. These are happening as our client tries to increase the rate but realizes that the server is skipping requests and thus slows down again, but in the process gets a lot of penalties. So we generally decrease the rate more cautiously.

III. **Testcases ran on localhost server** (tested on 100k Lines)

- Result: true
  Time: 27.865
  Penalty: 29
- Result: true
  Time: 28.908
  Penalty: 18
- Result: true
  Time: 28.651
  Penalty: 32

NOTE: with constant rate we were able to achieve time of 16.657 seconds with Penalty : 1.

e. Graphs:

We have plotted the following graphs to analyze the performance of our protocol in a variable rate environment :

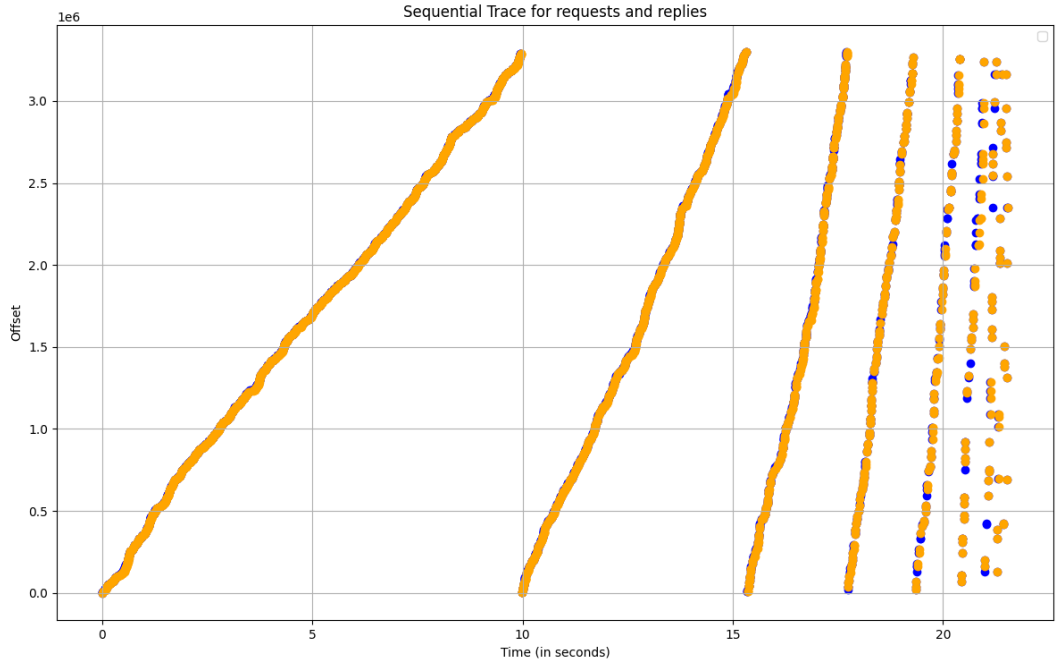

Figure 13: Sequential trace for receiving the file from Vayu server

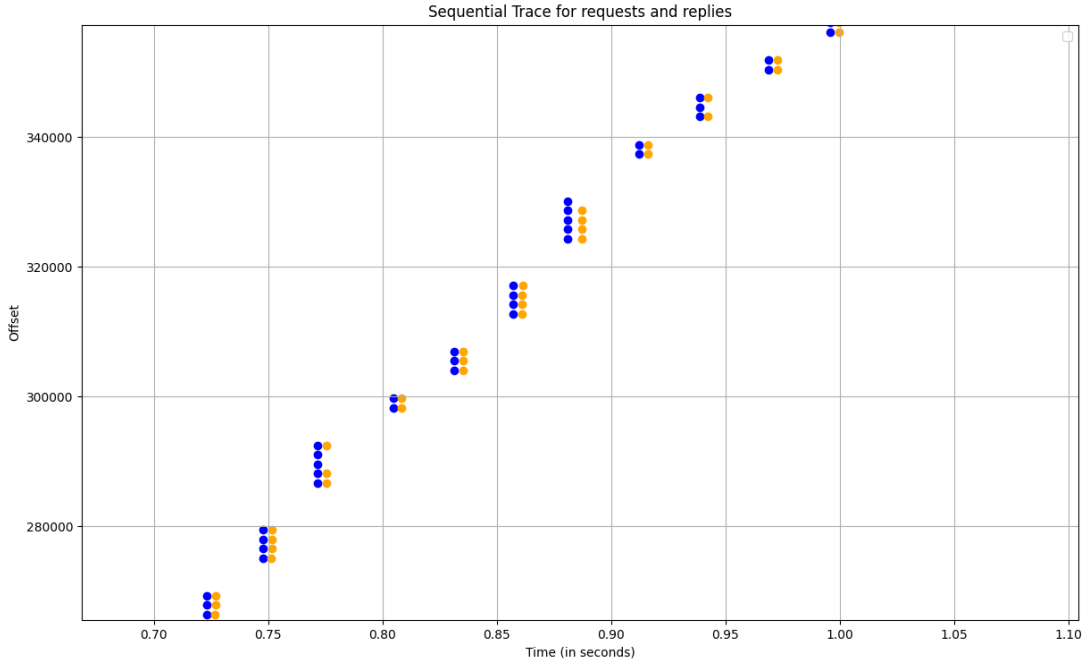

Figure 14: Sequential trace for receiving the file from Vayu server, Zoomed In

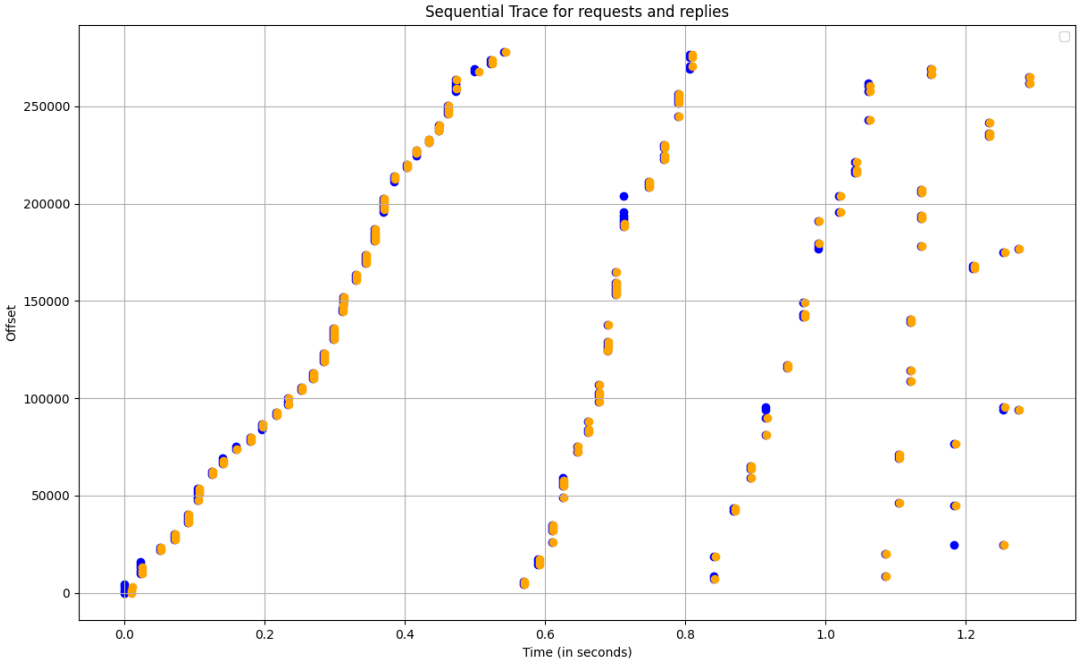

Figure 15: Sequential trace for receiving the file from Localhost server, 10K lines

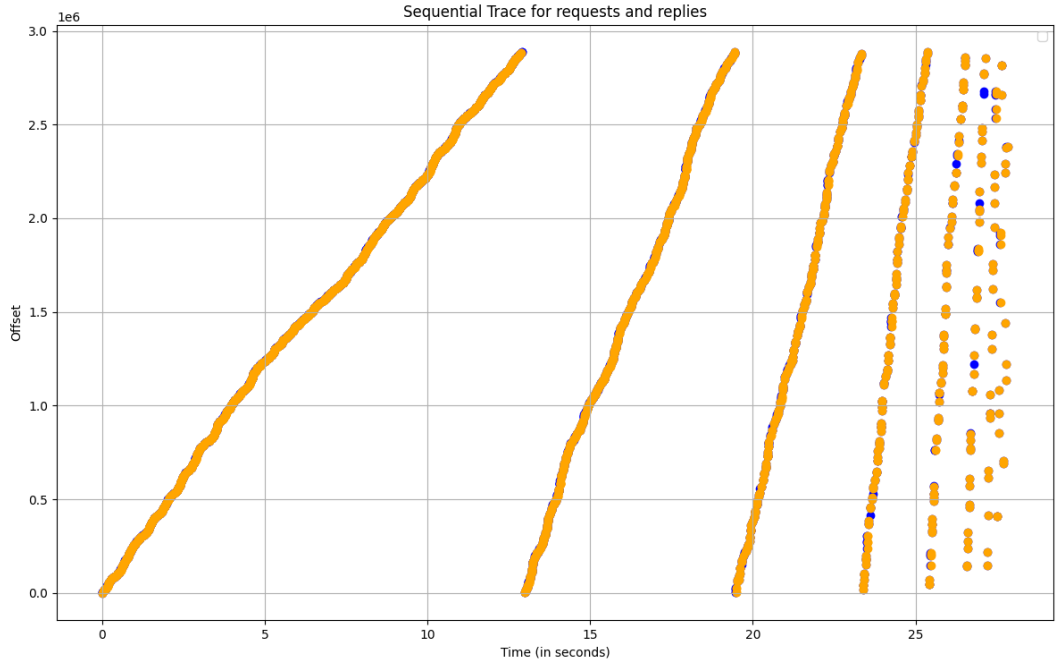

Figure 16: Sequential trace for receiving the file from Localhost server, 100K lines

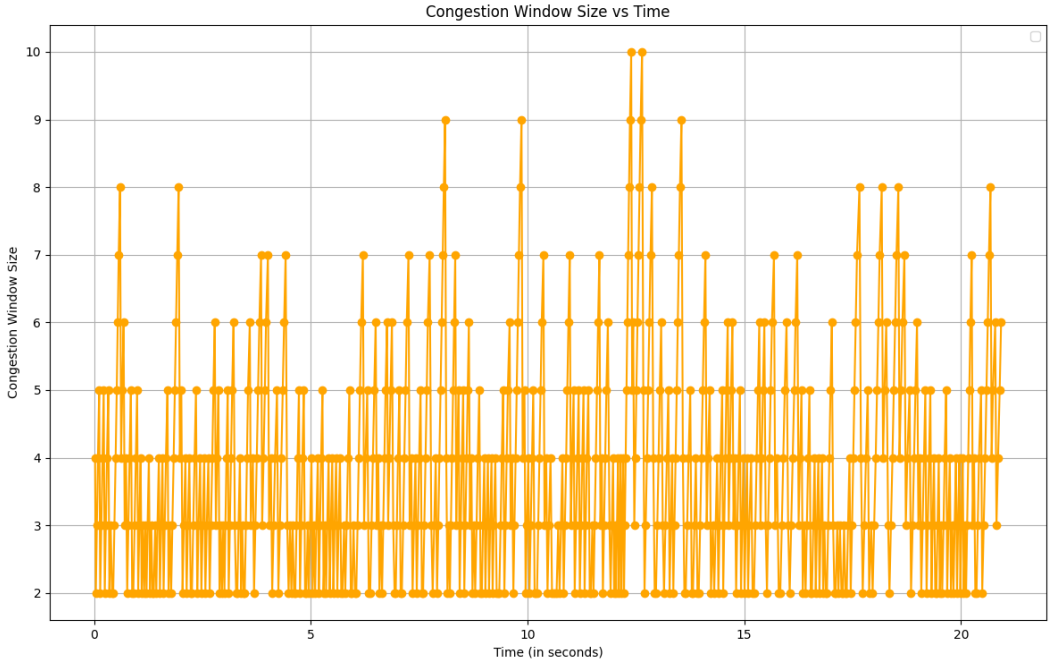

Figure 17: Burst size varying with time

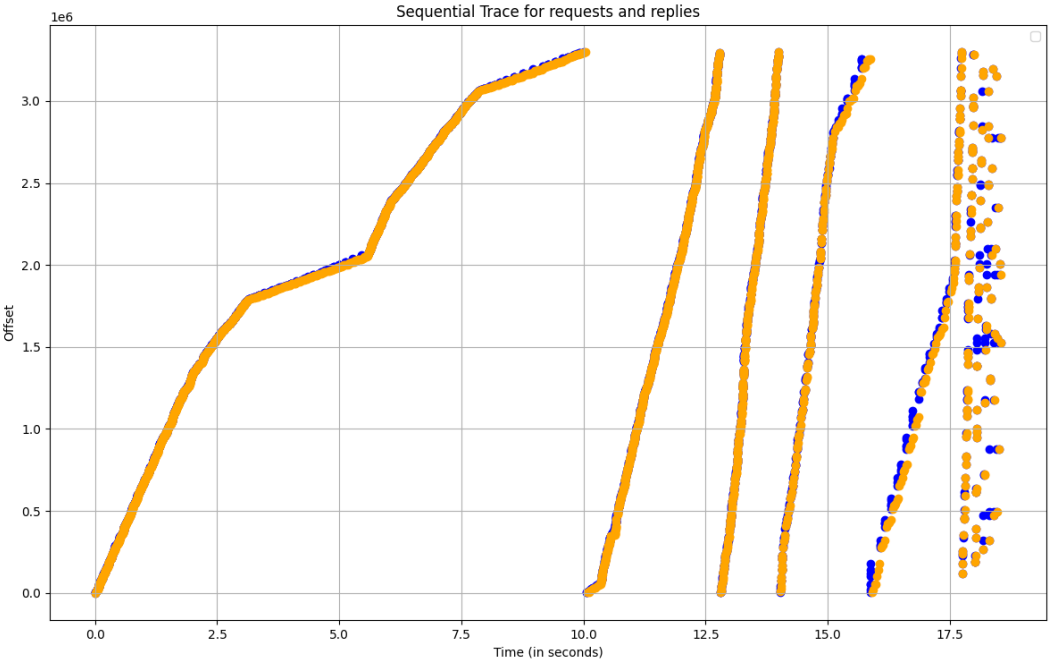

Figure 18: Sequential trace for offset vs time but also getting squished.

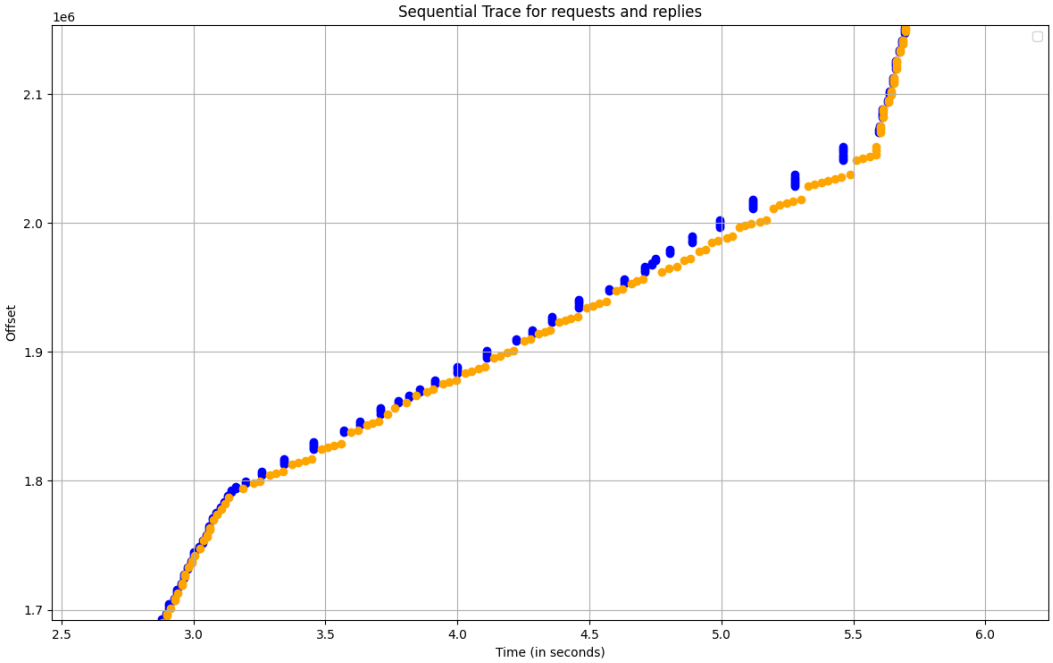

Figure 19: Sequential trace for offset vs time but also getting squished, Zoomed In

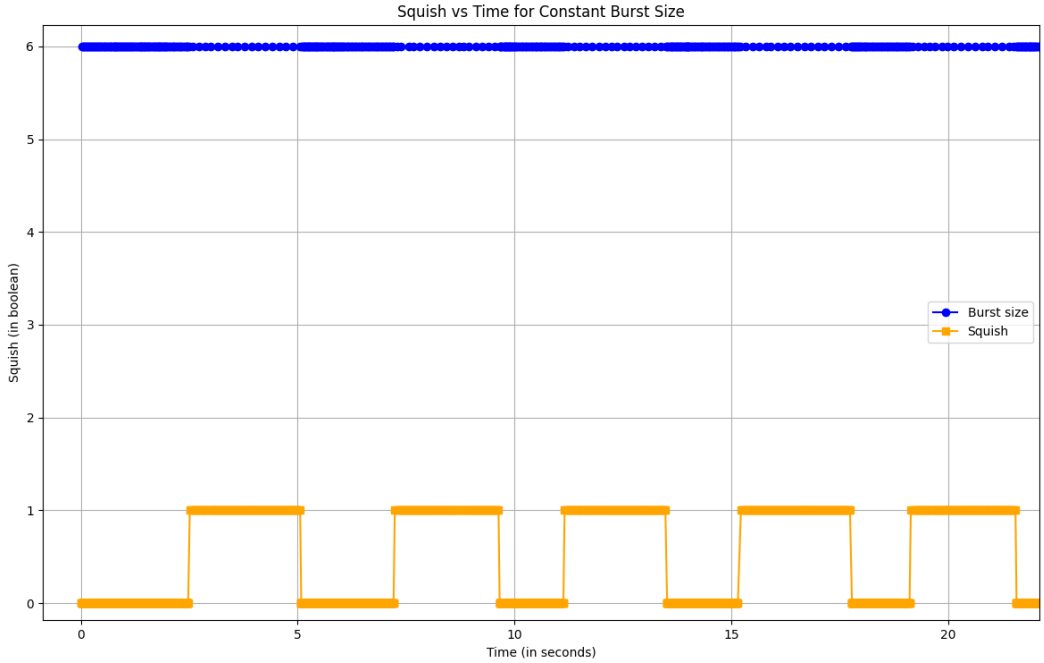

Figure 20: Getting squished while sending data at a constant burst

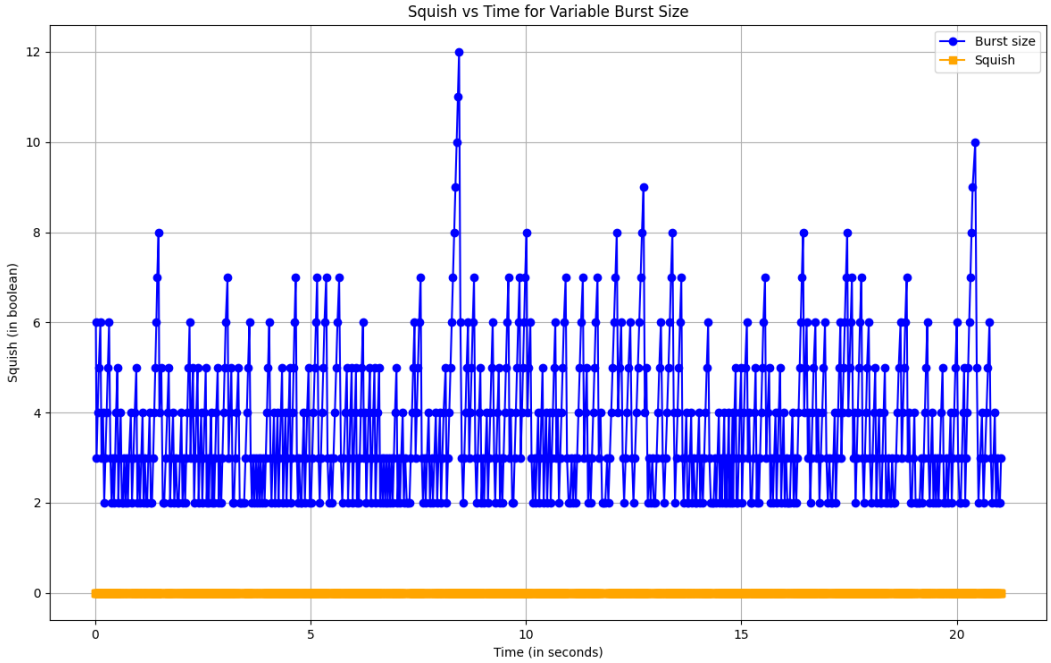

Figure 21: Sending data at a variable rate without getting squished

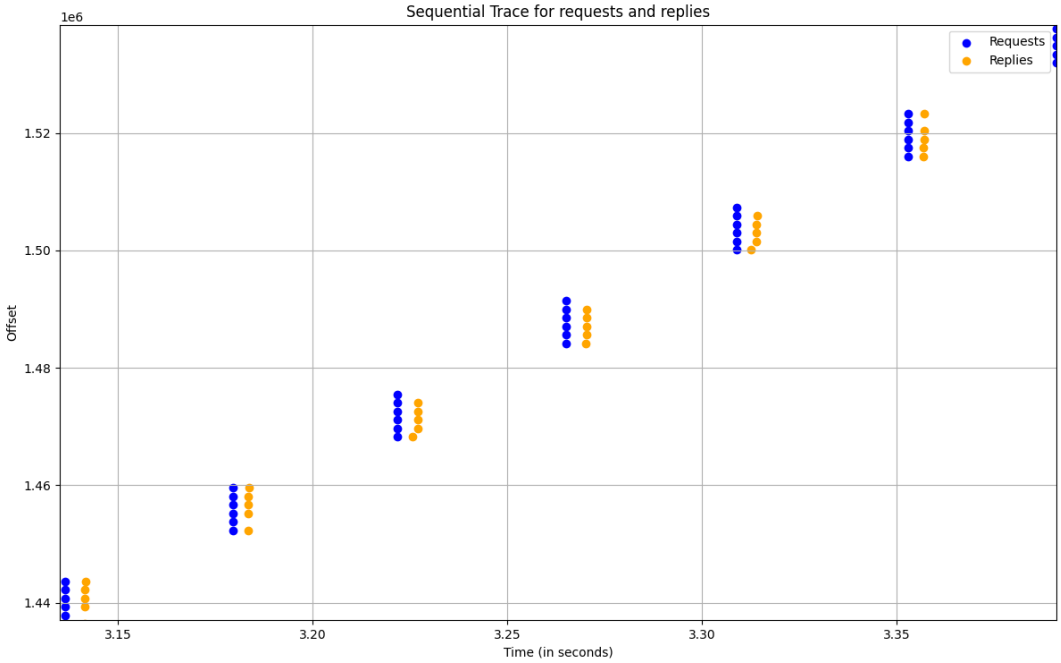

Figure 22: Sending data at a constant burst to Vayu server

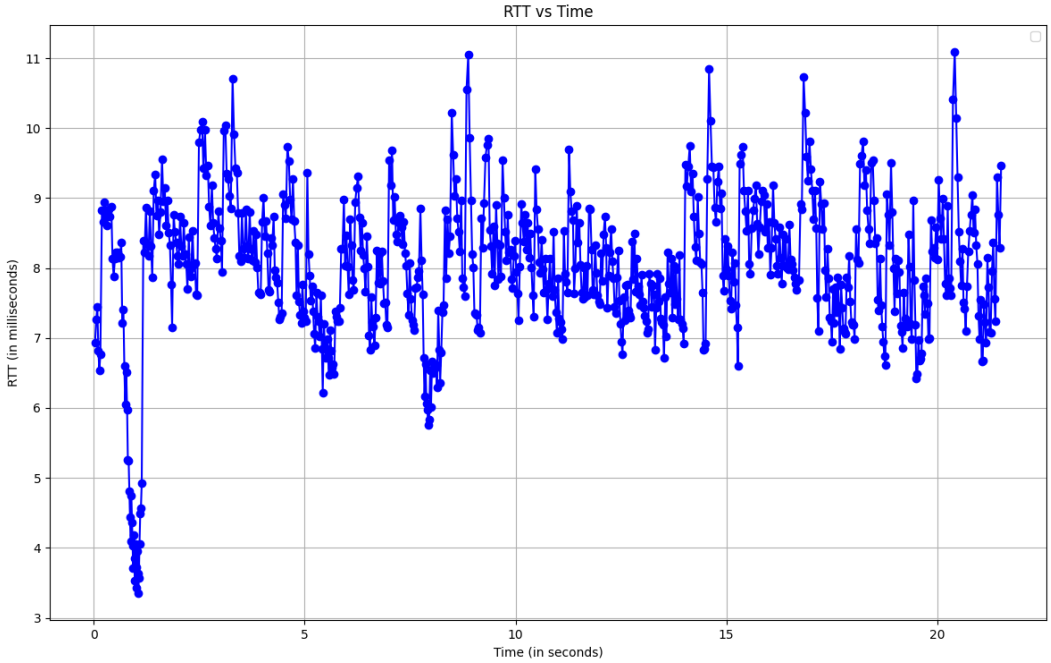

Figure 23: RTT varying with time (Sampled using EWMA equation)

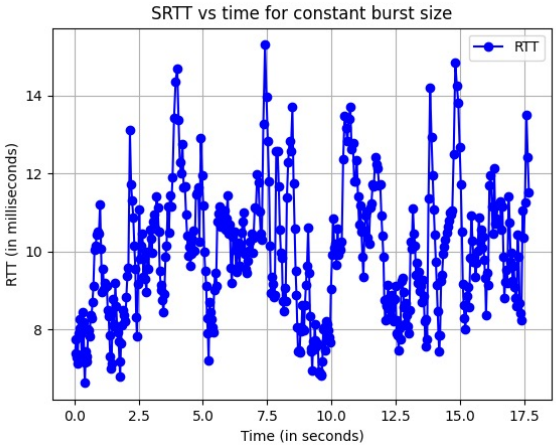

Figure 24: RTT vs time for Constant burst size

f. Comments on the Graphs:

- The first graph `Figure 13.` shows the offset vs time trace for receiving file from Vayu server. It clearly shows that our protocol requires multiple iterations to receive the whole file and after each iteration lesser packets are left. The graph keeps on getting sparse as lower and lower packets are left. The next figure `Figure 14.` just shows the graph zoomed in which shows burst sent and replies received after that.
- Similar to the previous graphs, we have also traced out graphs for the localhost server with both 10k lines and 100k lines `Figure 15.` `and Figure 16.`. In the 10,000 lines graph, the varying of the rate of

receiving packets can be seen more evidently.

- `Figure 17. and Figure 21.` shows the burst size sent varying with time. The burst size is set to a min value of 2, i.e., it never goes below 2 bursts.
- `Figure 18. and Figure 19.` shows the offset vs time graph for the case when the client was getting squished by the server. It is clear from the trace that the requests and replies slowed down during the squished period (`in Figure 19.`, the graph is much more sparse in the squished region). Also the squish periods can be seen with decreased slope in the graph.
- `Figure 20.` shows the client getting squished for the case when it doesn't adapt the burst size according to the server's variable rate.
- `Figure 23.` Finally shows the graph for the RTT varying with time. The RTT here varies a lot compared to the graph that was plotted in milestone 2. This is due to the fact that now we are sending bursts of varying sizes. This can be compared with the `Figure 24.` where RTT is plotted for constant burst size and varies a lot less

g. Configurable Parameters:

- **Timeout Period:** Timeout period defines the time our client waits for a reply before getting a socket timeout error. This was set roughly equal to the RTT.
- **Burst size** : This specifies the number of requests we will send in a burst. This is updated after every burst sent and received according to the server's rate using AIMD approach as explained in the protocol.
- **Freeze Time:** Freeze time is the amount of time our client waits after sending a request to allow the server to regenerate tokens and not overwhelm the server with rapid requests. We keep updating the freeze time in our protocol depending on the RTT using the EWMA equation (*equation*2).

h. Curious Observations:

- We observed that the time taken to receive the file varies significantly on the vayu.iitd.ac.in server and the other servers that were hosted at the time. The penalties incurred were similar, but the time differences were significant, like, it was taking 21-22 seconds on other servers, whereas on vayu server it was taking around 25-26 seconds on average.
- We also found out that the same code receiving file from the same server IP behaves quite differently on different machines (e.g. on Linux and Mac). We think that this is due to the network optimizations that the OS might implement on the machine.