Reliable Data Transfer

Team: joy_maa

Anish Banerjee* Ankit Mondal†

October-November 2023

Contents

1 Checkpoint 1 **2**
1.1 Our Method . . . 2
1.2 Graphs . . . 3
1.3 Appendix: Other Attempts . . . 4

2 Checkpoint 2 **5**
2.1 Our Method . . . 5
2.2 Graphs . . . 5

3 Checkpoint 3 **8**
3.1 Our Method . . . 8
3.2 Observations . . . 8
3.3 Observations from graphs . . . 9
3.3.1 Graph 1. . . . 9
3.3.2 Graph 2. . . . 9
3.3.3 Graph 3. . . . 9
3.3.4 Graph 4. . . . 10

4 Comparative Analysis **11**
4.1 Additive Increases in discrete steps v/s fractional increases . . . 11
4.2 To maintain, or not to maintain a Congestion Window? . . . 12
4.2.1 Observations . . . 13
4.3 Hardware Matters! . . . 14

*2021CS10134
†2021CS10229

§1. Checkpoint 1

1.1. Our Method

- To receive the data reliably, we maintain an `receivedlist` initialized with a dummy character. In each iteration, we update the `receivedlist` with the data received from the server. This helps us maintain the data's order, as it is not received sequentially from the server.
- However, the server may refuse to respond to some queries hence we do not expect to receive the entire file at the end of the first iteration. So, we repeat this process multiple times till we receive the entire file.
- We have also calibrated the timeout to a suitable value so that we don't wait too long for skipped requests.
- We have also added a wait period after every message to ensure that messages are sent at more or less uniform rate without resulting in squishing the data.

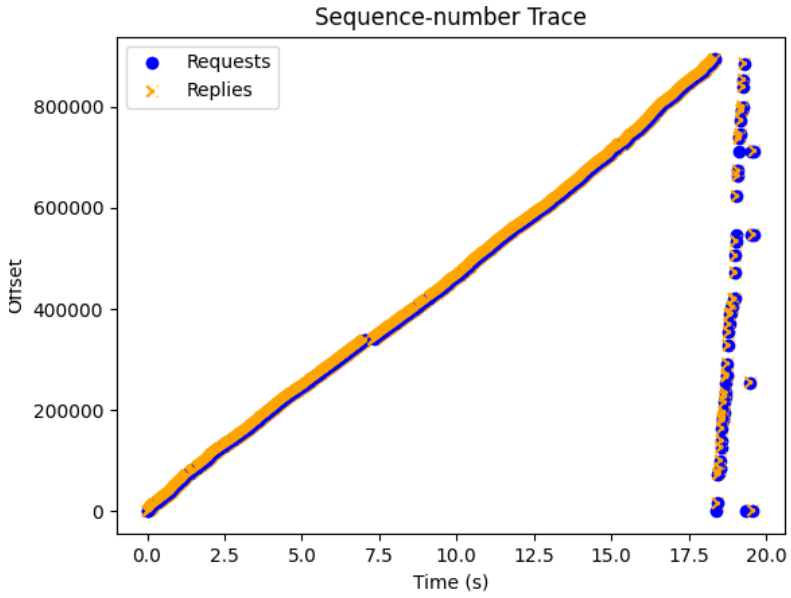

Figure 1: Sequence-number trace

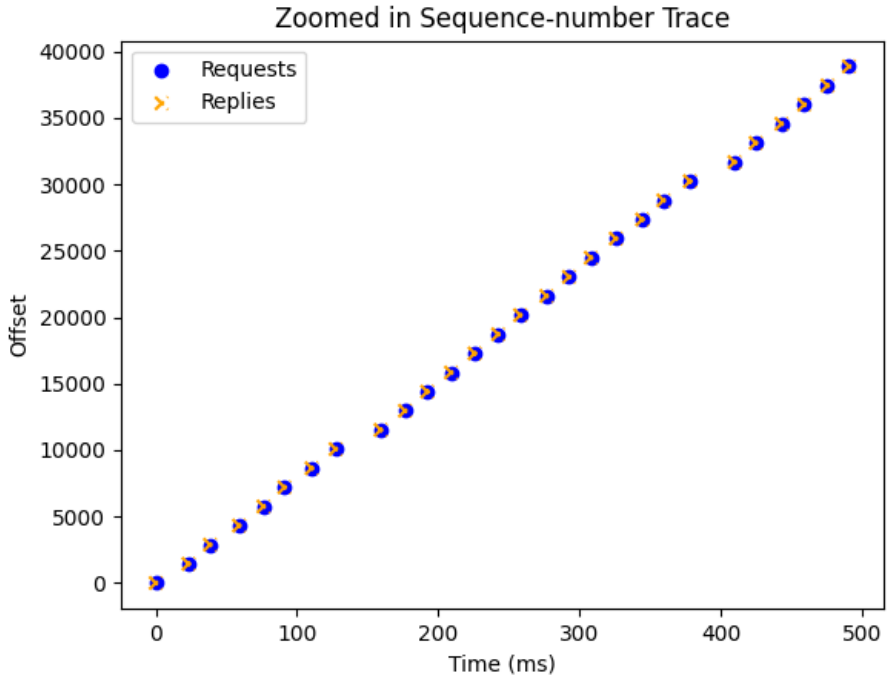

Figure 2: Zoomed in Sequence-number trace

1.2. Graphs

- Fig. 1 shows the sequence number trace for an approx 0.01 MB file. Requests are shown in blue and replies in orange. The replies appear to almost overlap with the requests. As mentioned in our method above, we make multiple passes to obtain the entire file. This is evident in the graph too - approximately,
  - The first pass starts at 0s and continues till 17.5s,
  - The second pass starts at 17.5s and ends at 18s and,
  - The last pass starts at 18s and ends almost immediately as it sends just 4 requests.
- Fig. 2 shows a zoomed-in version of the above trace for requests and replies within 500ms from the initial time. From this too, we observe that the requests and responses are almost overlapping. There is a very small round trip time and in a few messages **there is a visible gap between the query and reply**. As the RTT increases the gap between 2 queries also rises (as time between successive queries is round trip time + sleep time per message). Some **queries are dropped midway**. These queries are raised again in the subsequent passes.

1.3. Appendix: Other Attempts

We also tried implementing a stop-and-wait protocol in which the sender keeps sending requests till it receives a particular line. When a particular message is dropped, it keeps sending the same request till the response is received. We also tried sending the requests in burst sizes, decreasing the burst size if we had to wait to long for a particular message.

Result: The time to receive file decreased but penalty increased as some messages are squished while trying to find the correct rate to send the requests. Thus for the time being we move ahead with the safer code mentioned at the beginning of the report. The overall graph is also showing much less linearity as compared to previous uniform rate version.

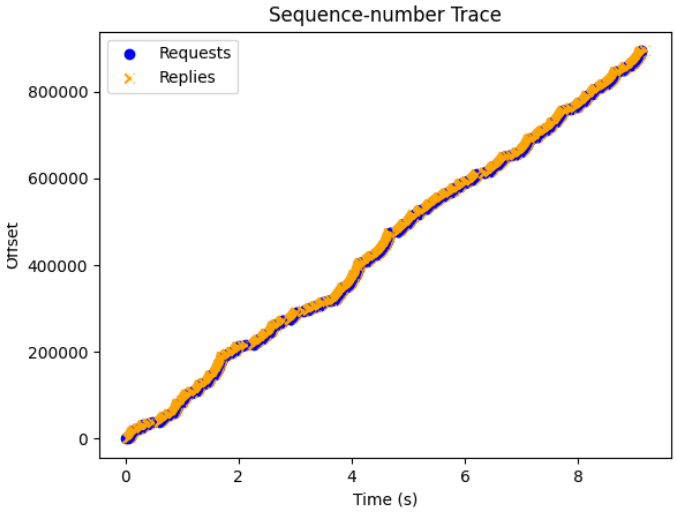

Figure 3: Sequence-number trace 2

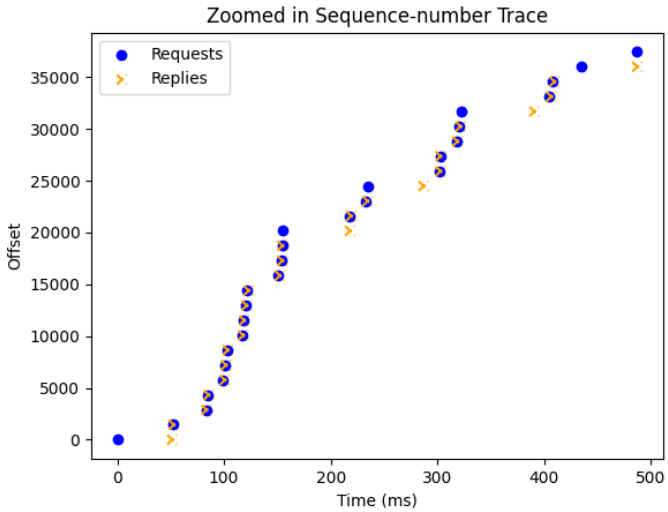

Figure 4: Zoomed Sequence Number Trace 2

§2. Checkpoint 2

2.1. Our Method

We made the following modifications to the earlier approach:

- **AIMD:** We send requests in bursts whose size is controlled by AIMD:
  - If we receive responses to all the requests, we increase the burst size $b = b + 1$
  - However, if the difference between the number of requests sent and the number of responses received is large, it means that we might be exceeding the number of tokens in the bucket. When this difference crosses a certain threshold (determined by the RTT), we increase the burst size by $b = b + \frac{1}{b}$ instead of 1. This helps to decrease the penalty.
  - If some requests are lost, or we get squished, we multiplicatively decrease the burst size $b = \frac{b}{2}$
- **Time between two bursts:** If the burst size is stuck at 1, it means that our rate of sending bursts is too high for the server, even if we send one burst at a time. So, we increase the sleep time between two bursts.
- **EWMA:** To measure the RTT, we use EWMA:

  $$\mathsf{RTT} = \alpha\mathsf{RTT} + (1 - \alpha)\mathsf{sampleRTT}$$

  where $\alpha = 0.125$
- **Optimal Sleep Time:** We performed a binary search for optimal sleep time on Vayu (≈0.008s). However, the same value did not work for Localhost, where the optimal value was ≈ 0.02s. The fractional increase approach helped us adapt our code to both situations since the burst size never increased too much.

2.2. Graphs

Fig. 5 shows the graphs obtained for the constant rate server running at Localhost and Fig. 6 shows the graphs obtained for Vayu 9801.

Observations

- **Total time taken** to obtain the file for Vayu is around **18s** while that for Localhost is around **25s**. The time for Localhost is more than that of Vayu because the file sizes differ (Localhost ≈ 1,00,000 lines; Vayu ≈ 50,000 lines).
- **Penalties** for Vayu is **9** while for Localhost it is **52**.
- In both cases, we require roughly 4 passes to obtain the entire file.
- The requests and responses almost overlap with each other in the graph for Localhost. On the other hand, the difference between sending and receiving is more visible in the Vayu graph. This is because the RTT for Vayu will be more than that for Localhost.
- The burst sizes are seen to increase when the responses for the requests are received. However, the burst size gets halved when the responses are not received.
- If more requests are not received, the time difference between bursts increases.
- In the graphs for burstsizes v/s time, we observe that the burst size decreases suddenly after a certain point (near the completion of the second pass). This is because we are traversing through the received list sequentially in blocks, and as most of the packets have been received, we don't send requests for them again.
- We don't get squished in either of the two cases
- Compared to the (hacky) fixed-rate version implemented in the appendix of the previous section where wait time was hand-fixed by us, the time taken now and penalties are slightly larger. This is because the code incurs some penalty while finding the right rate to send the messages.

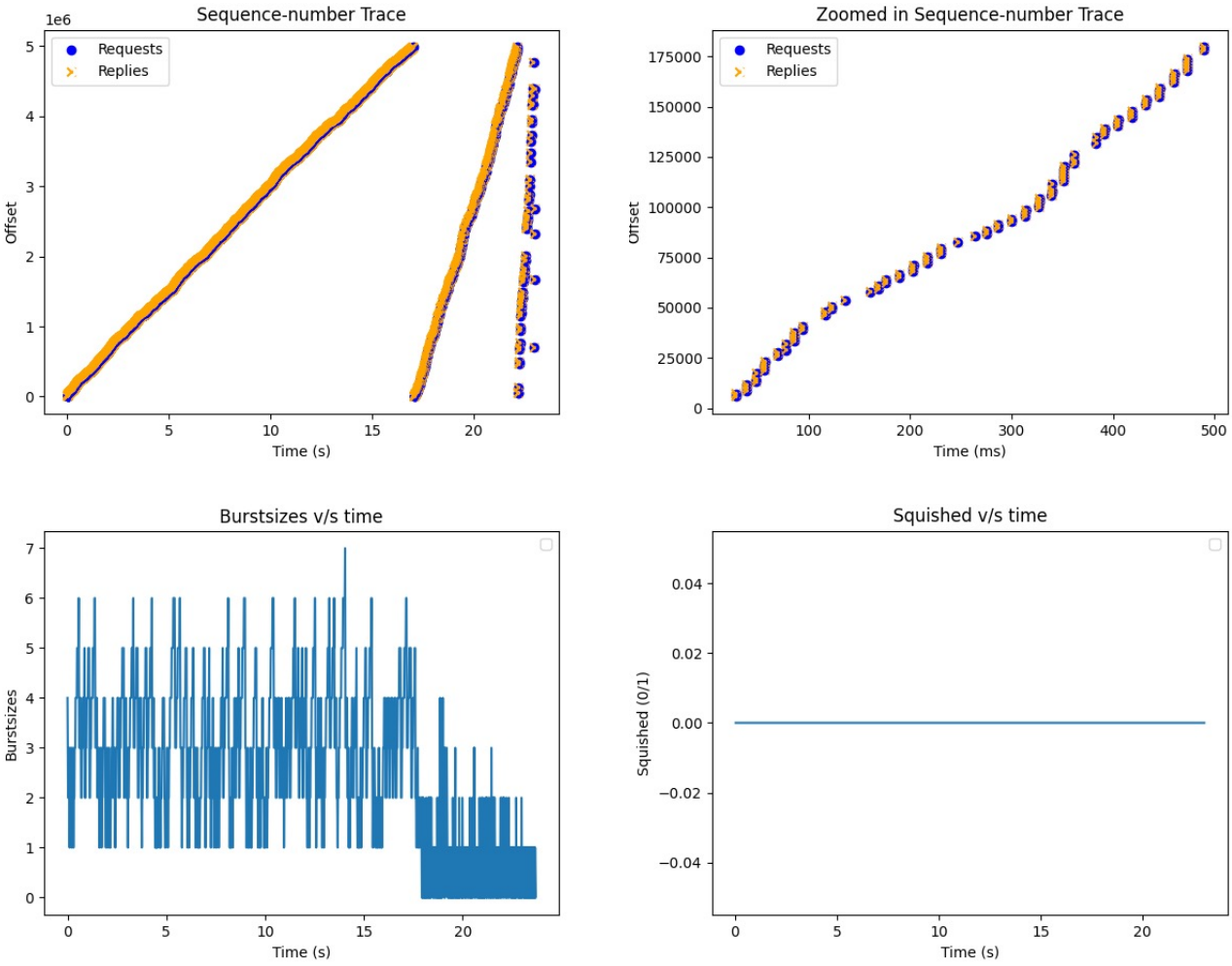

Figure 5: Graphs for Localhost

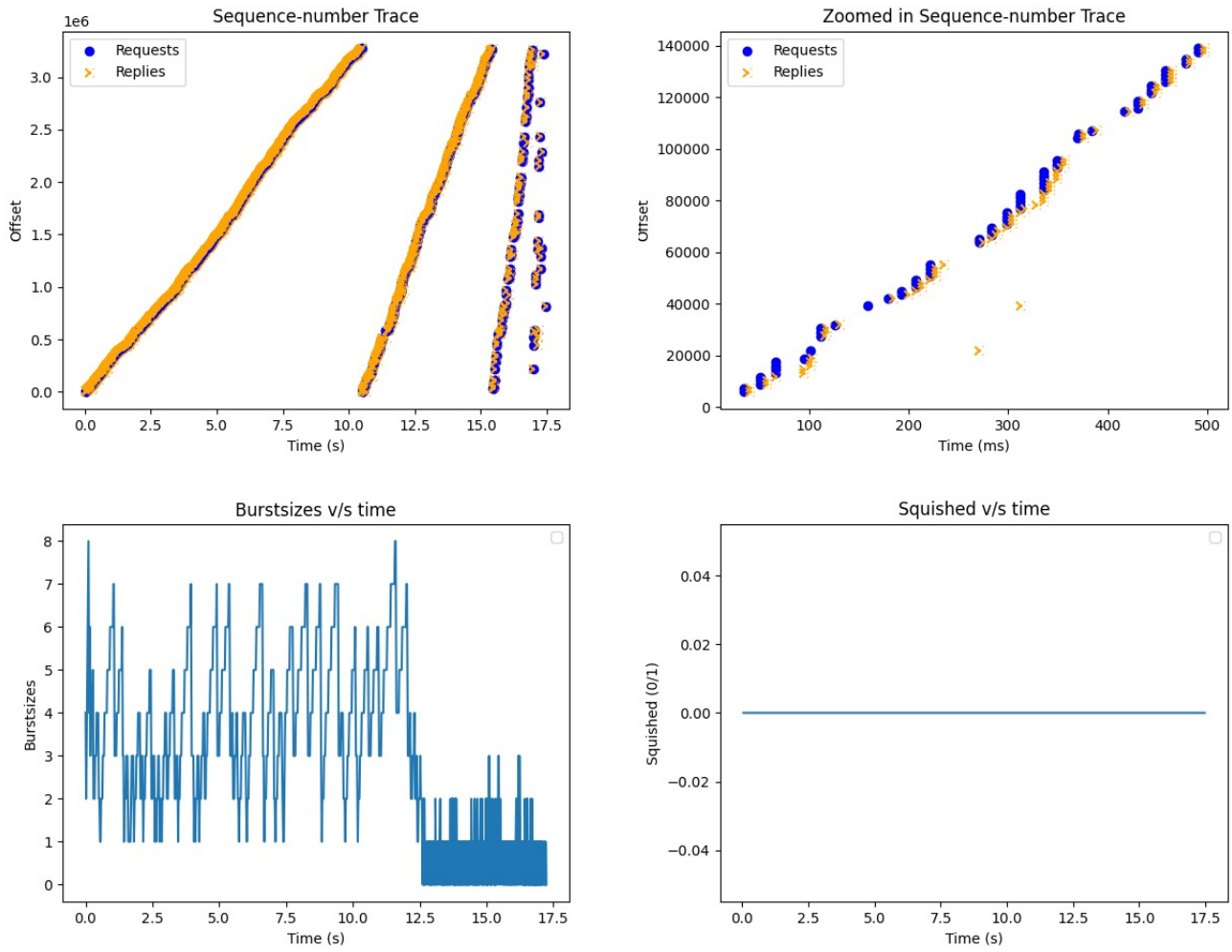

Figure 6: Graphs for Vayu

§3. Checkpoint 3

3.1. Our Method

Since in the previous checkpoint itself, our code was suited to variable rate servers, we have kept the logic the same. We also tried implementing a separate code to learn the RTT, which was giving worse results (check Section 4.2). One new addition to our protocol is the pre-calculation of RTT:

- We are sending `SendSize\n` requests to learn the RTT first.
- In a `for` loop, we send ten `SendSize\n` requests and we record the difference between the time of sending the request and receiving them.
- We then take an average of these times to estimate the RTT.
- Then we send requests for the file using the calculated RTT.

We chose this approach because on setting the RTT according to EWMA on the fly, we noticed a large variation in the recorded RTT times. This is because, in the burst size-based algorithm, we are waiting for some sleeptime after every burst. If a message is waiting in our receiving buffer, this sleeptime gets added to its round trip time. Even though it has already reached the receiving computer.

Example Suppose a message m enters the buffer at 10s. It was requested at 9.9s, so actual RTT is 0.1s. At this stage the program is reading k (burst-size) number of requests from the buffer. Before m is read, the buffer finishes reading k requests and starts sleeping for 20 milliseconds. After it has sent the next burst (this takes minimal time) it again starts reading responses and now it reads the message m. The process of reading by itself adds very little extra time to the rtt, but the extra 20 milliseconds adds a considerable additional time to the rtt. Since, we have kept timeout of the socket to rtt, this additions ends up compounding over time. If the network has a slow rate then this approach gets overtly cautious and waits for a very long period at times (when the packet is randomly dropped).

3.2. Observations

Here is a table of our experiments with various servers on port 9802. These are the results for running on Lenovo Ideapad Windows 11 with intel i5 processor. On running on a Mac, we got an average time of 22s (check Section 4.3)

Server (Port 9802)	**Time (s)**	**Penalty**
`10.17.7.218`	27.619	34
`10.17.7.218`	28.990	48
`10.17.51.115`	26.734	39
`10.17.51.115`	28.524	48
`vayu.iitd.ac.in`	27.288	7
`vayu.iitd.ac.in`	28.8857	13
`localhost`	27.369	20
`localhost`	27.798	28

Table 1: Time and Penalty across different servers

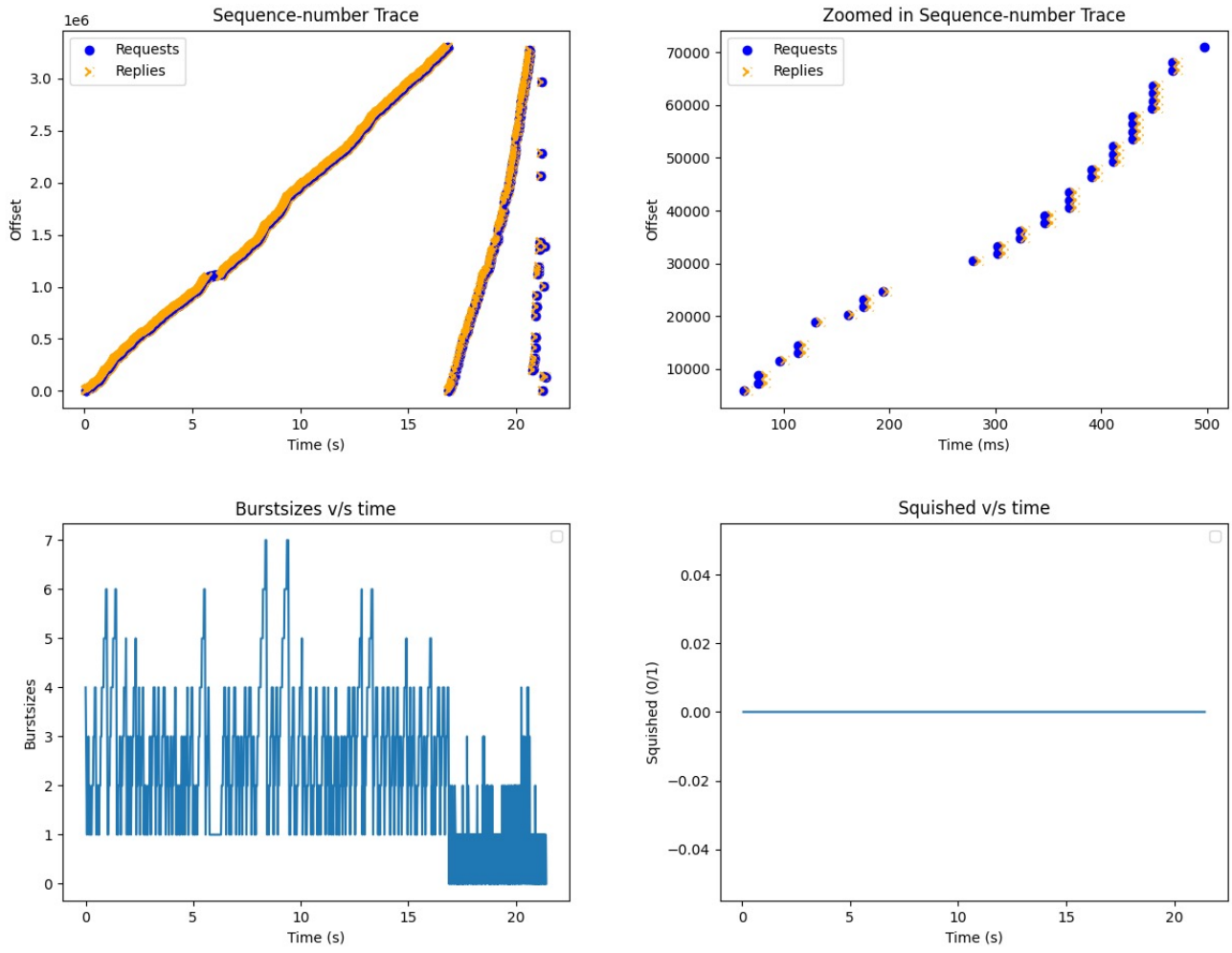

Figure 7: Graphs for Vayu

3.3. Observations from graphs

3.3.1. Graph 1.

- It takes upto 4 runs for the entire file to be received at the receiver.
- The graph for each run is not linear. There are abrupt changes in the slope of the graph in the first run which signifies that the rate was variable for the duration of receiving the file

3.3.2. Graph 2.

- The different bursts are sufficiently spaced.
- There is a gap in the bursts which is because of a dropped packet which was received in the second run.
- The increase in burst sizes is not immediate as the increase is fractional.

3.3.3. Graph 3.

- The variation in burst sizes is much more as compared to the constant rate server. This the reason why the graph is much denser than the old constant rate graph.

- There is also a considerable variation in the maximum burst size reached. The average max-burst size is 4 but some 5,6,7 sized bursts were also seen. In constant rate the max-rate was almost saturated at 7.
- Some gaps can also be seen where the burst size is predominantly 1, signifying that rate was very low at this time and some bursts sizes were minimum that is 1.

3.3.4. Graph 4.

- Since we are taking a cautious approach, we are not getting any squishes for our run.

The following graph represents our estimated RTTs using EWMA:

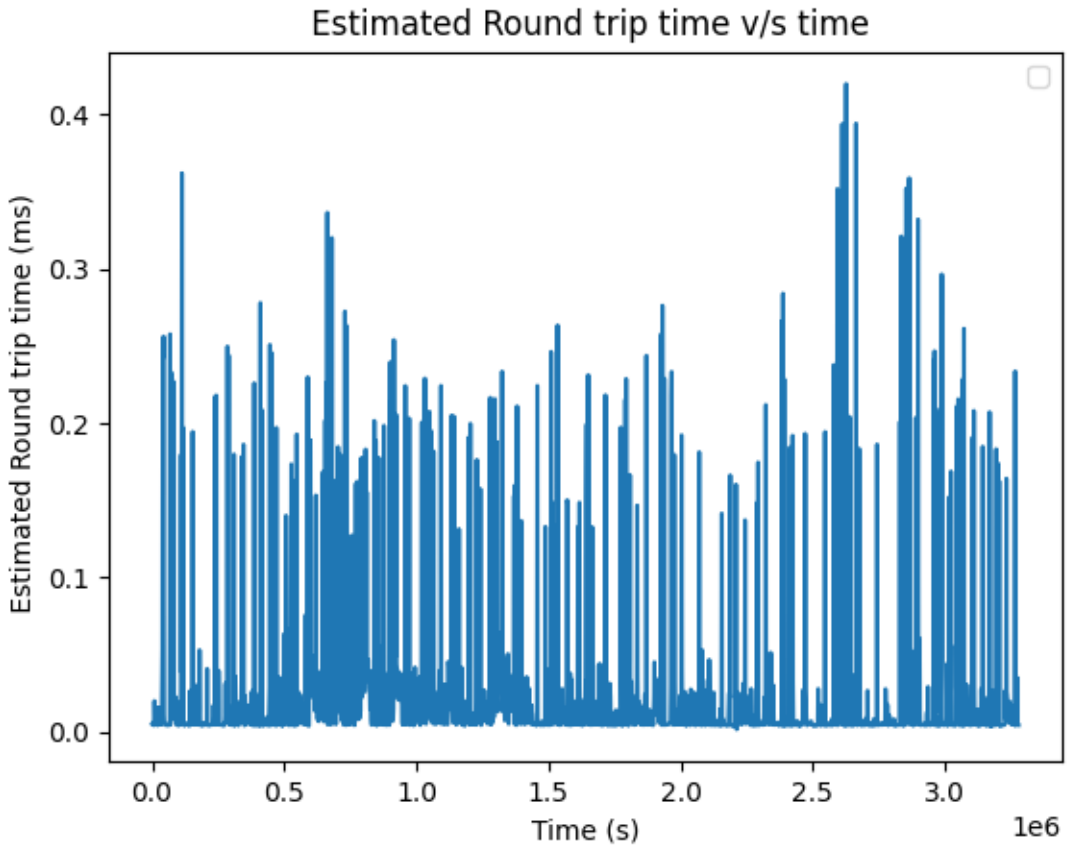

Figure 8: Estimated RTTs wrt time

§4. Comparative Analysis

4.1. Additive Increases in discrete steps v/s fractional increases

Table 2 sumarizes the results for increases in discrete steps and increases in fractional steps.

Approach	Server	Time (s)	Penalty
Discrete increases	`vayu`	26.528	64
	`10.17.51.115`	17.486	111
	`10.17.7.218`	21.208	183
Fractional Increases	`vayu`	27.884	14
	`10.17.51.115`	22.926	29
	`10.17.7.218`	23.671	20

Table 2: Time and Penalty for fractional and discrete increases3

From the above table, we can conclude that

- Fractional increases by 1/(burst size) are observed to result in lesser penalties than discrete increases.
- **Example:** Suppose we have a burst size of 5 and the number of tokens in the bucket is 6. Then, with a discrete increase, we will send a burst of size 6; for the fractional increase, we will send a burst of 5. The second case has a penalty of 4, while the first gets a penalty of 5.
- In the above table, we see that the penalty for discrete increases is significantly higher than that for fractional increases. It even gets squished twice.
- However, the time taken by the additive increases is somewhat less than that for fractional increases. This is expected since the burst sizes increase more rapidly for discrete increases.

The following graphs also resonate with the above observations:

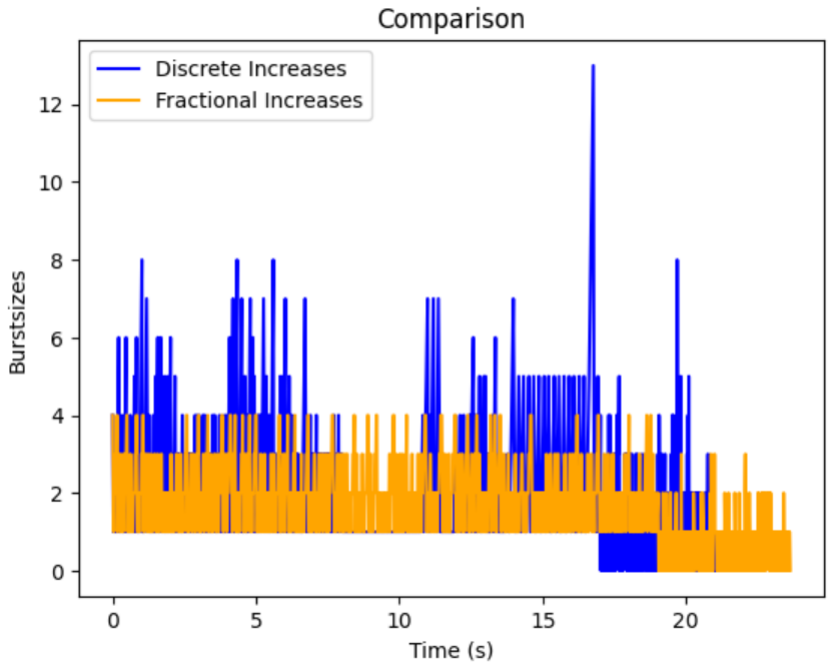

Figure 9: Comparative analysis of burst sizes for discrete and fractional increases

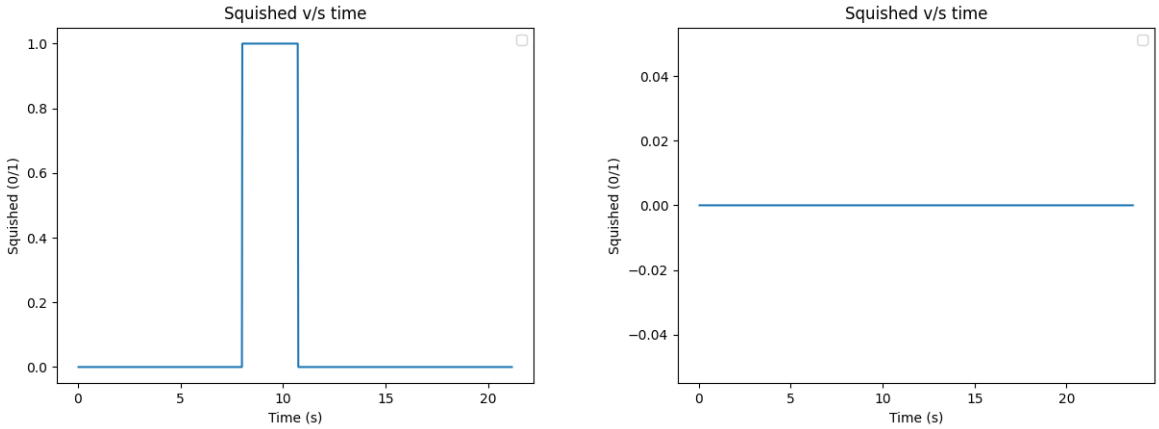

Figure 10: Comparison of squish for discrete and fractional increases

4.2. To maintain, or not to maintain a Congestion Window?

We observed that implementing a fixed-size congestion window gives worse results than our current implementation of AIMD (sending in bursts). In our congestion-window implementation:

1. We maintain a fixed size window of size $N = 10$, where we always have 10 packets in transit.
2. When we receive a response to a message, we send the request for the next message and shift the congestion window.
3. If we don't receive a response to our message at the beginning of the congestion window for 3 consecutive wait times, we resend a request for it.
4. We expect the sending rate to be roughly proportional to $1/RTT$. This is because we send a message only after receiving a response. So the duration between two requests is approximately equal to RTT.

Here are the graphs for this implementation

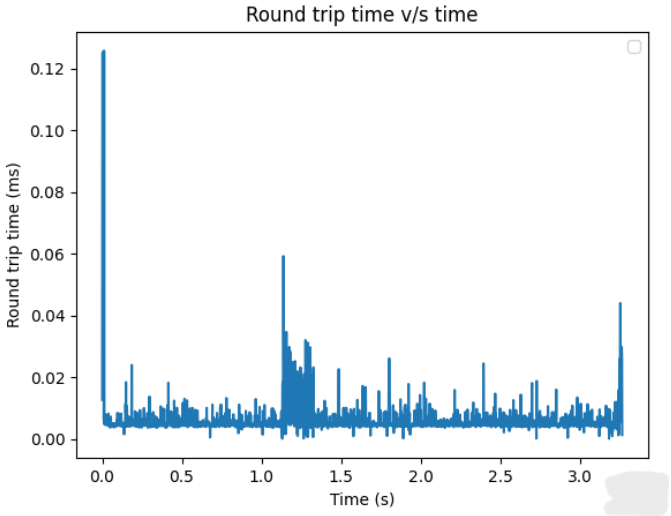

Figure 11: Roundtrip times v/s time

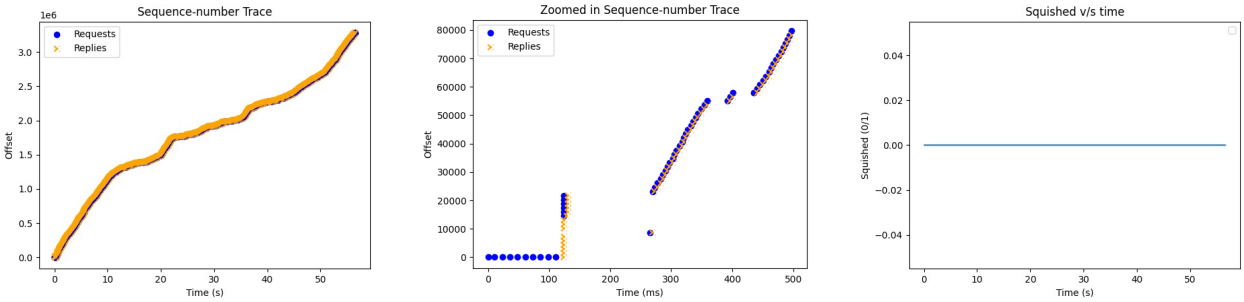

Figure 12: Graphs for Vayu

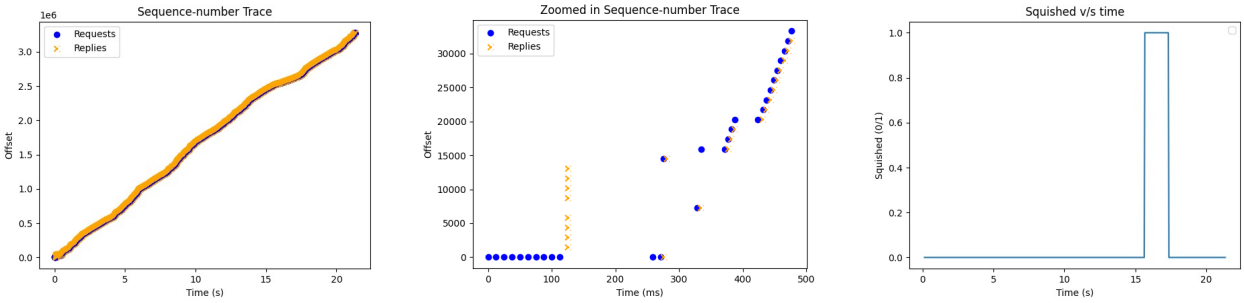

Figure 13: Graphs for 10.17.7.218

4.2.1. Observations

- On different servers, the behaviour is quite different. For `vayu`, the time taken is around 60s while for `10.17.7.218`, it is around 25s. Penalties too differ, for it gets squished in the latter but not in the former. The reason for this is the varying RTT of different servers.
- The rate of sending requests is proportional to 1/RTT since we send a new request only after receiving a previous one (assuming no losses).
- The penalties for this implementation will be high because the rate of generation of tokens will not necessarily be related to the RTT (our sending rate is 1/RTT).

4.3. Hardware Matters!

One interesting phenomenon we observed was that time taken and penalty taken changes with the system on which we run the code.

- One reason for this might be that the clock times of each system is differently synchronized. So the sleep time and timeouts might be different on different systems.
- Also, depending on the network cards the speed of accessing the packets from the lower layers of the network might be different, resulting in varying results.

Server	**Macbook**		**Windows**	
`vayu`	27.884	14	32.385	35
`10.17.7.218`	23.671	20	27.619	34
`10.17.51.115`	22.926	29	26.734	39
`localhost`	22.281	82	27.369	20

Table 3: Across Platforms

Considerable variation was also observed with the time at which the code was being run. The average time was around 22s on Mac and 26s on Windows. In very bad runs, time went up to 27s on Mac and 32s on Windows.