Assignment 3 - Milestone 3

Bhupesh Astha Meena

November 1, 2023

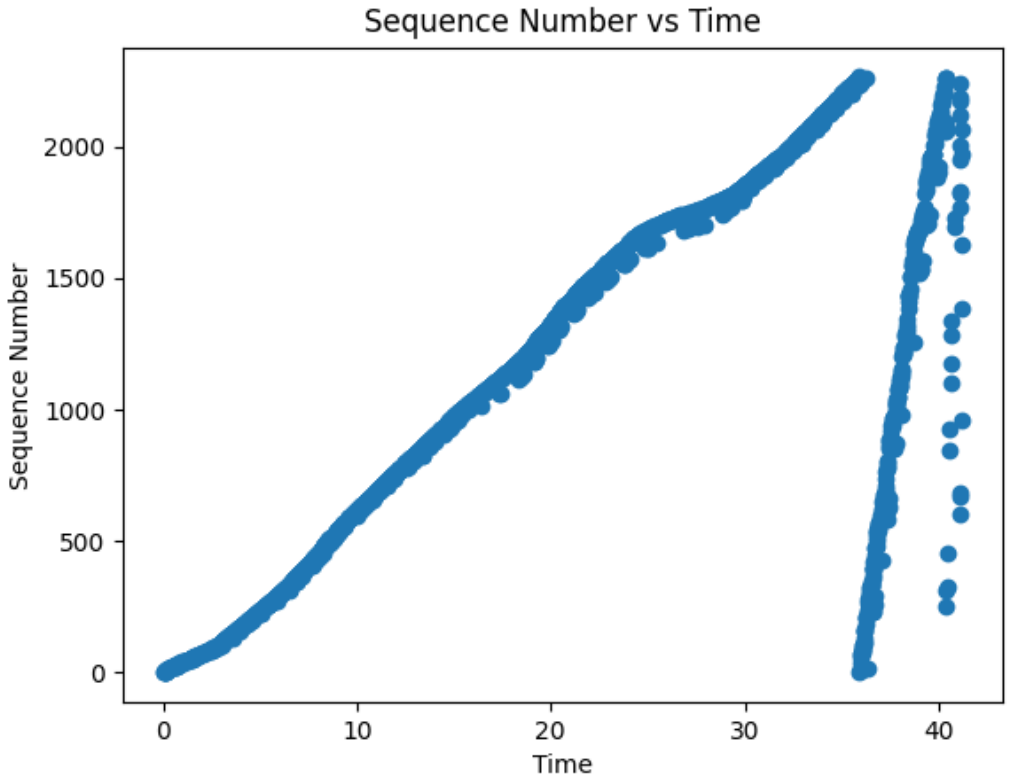
1 Code Working

- We get the size of file we have to receive by the SendSize command.
- We then calculate the number of requests to be sent by taking the ceiling of size/1448 as we are sending requests of size 1448 except the last. We define this variable as count.
- We then start a parallel thread to receive requests. We set a timeout for receiving requests so that the thread does not stop to receive only 1 request.
- We differentiate each request from their offset/1448 hash value. We store the data from each request sequentially in a list named full text based on this hash value.
- We are using the AIMD algorithm on the rate at which the client sends lines instead of the burst size. This sends lines continuously instead of bursts. We observed better results with this implementation.
- We initialize the sending rate with 100, and round trip time 0.01 and then we start sending requests . The
- If we get a message within the expected time (RTT) and it's not squished, we increase the sending rate additively.
- However, if the received message is squished, we reduce the rate by a factor of 2. We also pause sending requests for 1 second. We keep track of the time it takes to receive the data and update the RTT based on the best estimate, making sure it's not less than a minimum value of 0.0001 seconds i.e maximum(minimum(RTT,time2-time1),0.0001) . Finally, we add the received data to our collection of messages in full text.
- Our practical tests showed that a sending rate of 100 to 200 works best. This balance lets us send data efficiently while staying within our limits.

- We use additive increase to our rate whenever the losses are within the general threshold value we maintain by using the correct window variable. We use multiplicative decrease to our rate at /2 whenever the losses cross the given threshold.
- We have implemented a logic to check the number of responses and losses. We estimate this percentage using correct window and check if this ratio is maintained. We increase and decrease the rate linearly depending on increase in losses or decrease in them. This may prevent the data from getting squished by decreasing when the number of losses increase.

2 Graphs

2.1 Graphs using AIMD Algo on Sending Rate for Vayu Server



The graph depicting when the request was received with respect to the index of the request.

Observations

- The increae in sequence number is not static but ever changing due to the variable rate of the server sending lines.
- We need to send requests multiple times to recieve all lines , in the shown graph we, got all lines in three iterations. It can vary from 2,3,4,5..
- This is the graph when there is no squishing and minimal penalties are incurred.

2.2 Sequential trace vs time in a zoomed in version.

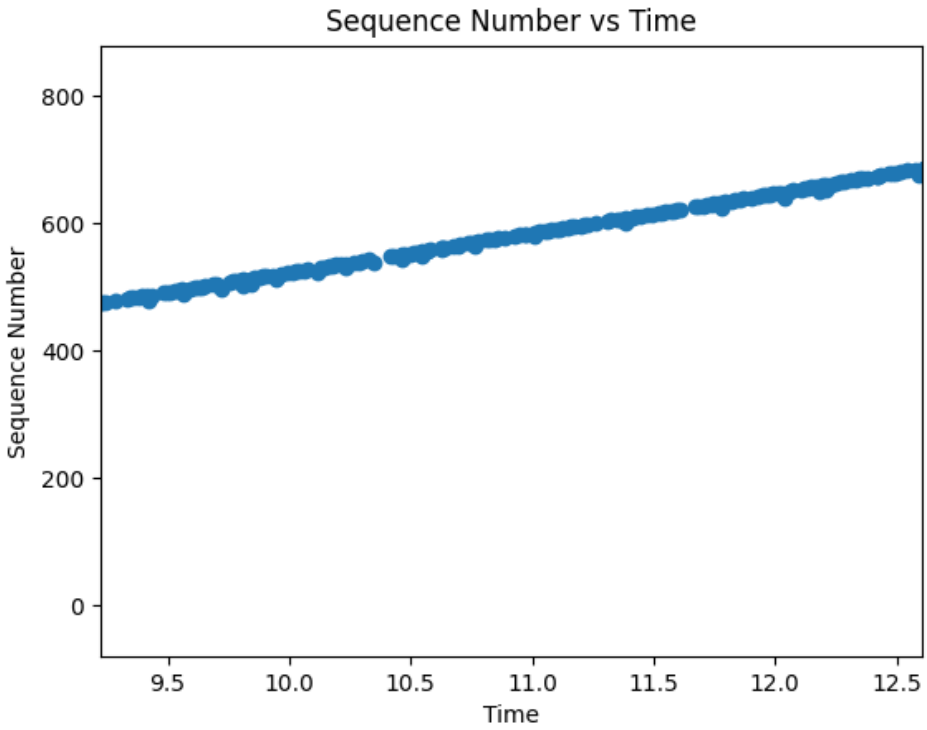


Figure 1: Zoomed in at the start

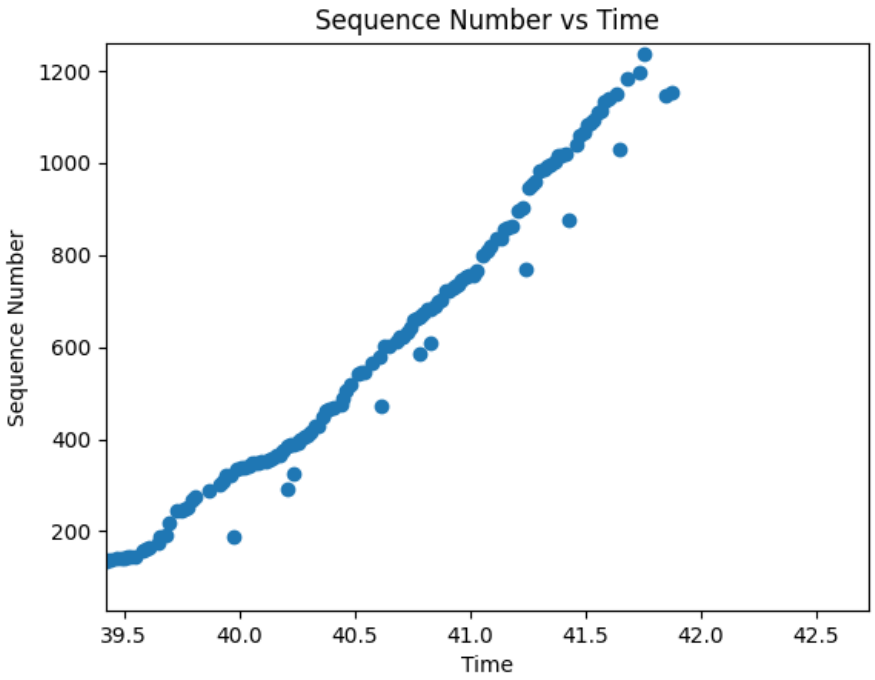
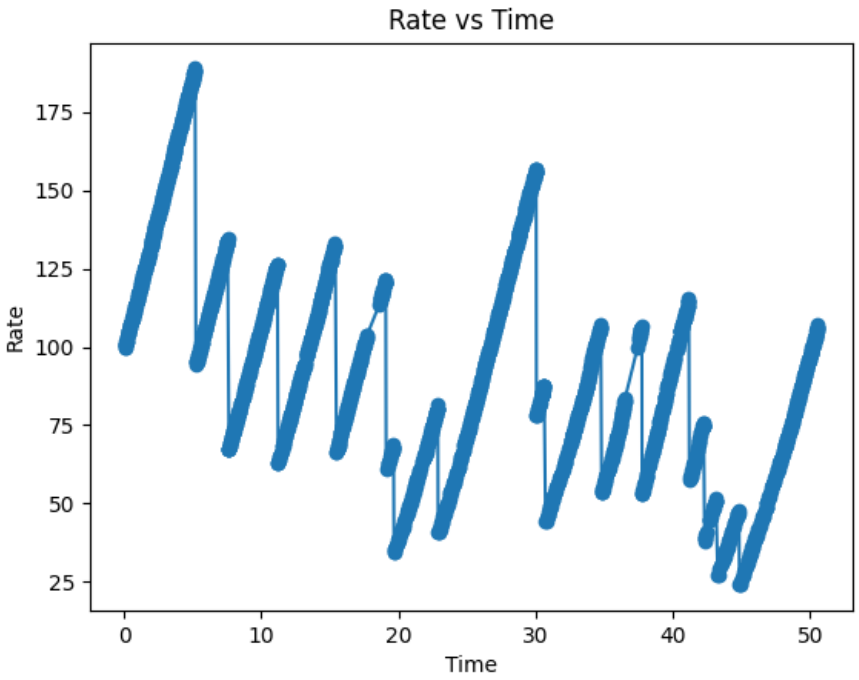


Figure 2: Zoom in at the end

Observations

- Here we can see the data is indeed not linear but changing after set intervals. The rate at which we get requests keeps increasing or decreasing.
- In the Zoom in at the start (Figure 1), the rate increase is constant because we start slow and the server rate is not reached.
- In the Zoom in the end graph(Figure 2), there are vertical gaps before the rate decreases generally. This is due to the server servicing fewer requests when the number of requests increases and our program responds to that and decreases the rate.
- This type of rate decrease generally happens in the later stages when the rate has already reached the maximum one allowed by the server.



Rate change in the graph

Observations

- The rate increases for some time interval and then abruptly falls as the rate of server sending the lines keeps changing in the server many times.
- The rate of the server is inversely proportional to the time we pause the sending code after each transmitted message. This approach ensures a steady stream of messages instead of transmitting them in bursts.
- The rate behaves differently compared to the case where we were gradually go towards a specific value of rate

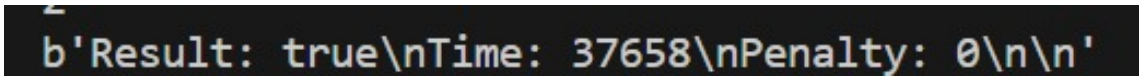
3 Result after sending MD5 hash to the server.

3.1 Sending at a faster rate and getting squished

```
b'Result: true\nTime: 34536\nPenalty: 158\n\n'
```

We can also get squished once or twice in our implementation if we start at a faster rate. The squishing happens due to the high initial rate and we get a high penalty in this case.

3.2 Sending at a Reasonable rate and not getting squished



- We get the result as true only due to receiving data reliably as we did in milestone 1.
- We also minimize the penalty to less than 50 in all cases and single digit in general by getting data reliably and not getting squished.
- The above image is the Result we get for the vayu server. We get the data in as less as 50 seconds constantly and only get a penalty of 0.