Computer Networks

Assignment3 : Milestone3

by Mani Sarthak & Harshit Gupta

November 1, 2023

Some points to be considered:

- All the codes are managed on **GitHub** and can also be found in this **folder**.
- Report of milestone 1 can be found **here**.
- Report of milestone 2 can be found **here**.
- For milestone 3 we had applied the following strategies to get the data.
  - Sending and receiving data in **bursts**.
  - The waiting time between adjacent bursts have been estimated using **EWMA** algorithm.
  - The final implementation is using **AIMD** algorithm.
  - Both sets of codes are prepared that have **AIMD** with and without constant size waiting window.
- **Parameter tuning** for optimisation have been tried out in the earlier milestones and now it have been automated in the script itself.
- Large data graphs were plotted using **matplotlib** library and rest using **LaTeX**.

Token Distribution:
1. Mani Sarthak : 2021CS10095 : 10
2. Harshit Gupta : 2021CS10552 : 10

Contents

§1 Logic Overview

The algorithms used are **AIMD (Additive increase multiplicative decrease)** and **EWMA(Exponentially weighted moving average)**. The scripts are prepared for both the scenarios in which AIMD is integrated with EWMA, and one in which AIMD works with constant window waiting size. We are sending and receiving the requests in **bursts**. As per the AIMD algorithm we increment the **cwnd** by 1 (additive increase) each time we receive all the replies corresponding to the requests sent, and decrease the **cwnd** by half (multiplicate decrease) otherwise.

§2 Experiments

1. **Experiment 1**
   - On observation for fixed waiting size window, and then increase the sending size to estimate RTT, we observed various patterns for the parameters: ***mean, min(RTT)/mean, max(RTT)/mean, median***.We concluded the formula for RTT based on several estimations and it comes out to be:-

     $$timeout = 1.2 * mean\_rtt + media\_rtt$$

     the penalties are quiet low and the implemntation is also way too fast. The 1.2 factor is purely observational but taking the median of the RTT's is a standard way.
   - Sometimes when the server is very clogged then this implementation have the mean RTT very high and hence the timeout is also set way too much than it should be. So we also put an upperbound of 0.02 seconds. Again this upperbound is also experimental.

#requests	Mean RTT	Min RTT/Mean	Max RTT/Mean	Median
20	0.0001729	0.8316	2.8932	0.0001531
40	0.0002414	0.5709	10.7583	0.0001462
60	0.0002246	0.4362	8.7121	0.0001721
80	0.0002049	0.4143	12.8814	0.0001149
100	0.0001852	0.4466	12.5510	0.0001521
120	0.0001736	0.4902	9.3366	0.0001290
140	0.0002084	0.4165	14.4172	0.0001478

Table 1 Data through which the formula for timeout was calculated

2. **Experiment 2**
   - In this experiment we implemented **EWMA** algorithm
   - we first send **SendSize** 50 times to get a set of RTT's, then we calculated median of them and set it as initial RTT
   - then when we send the burst and recieve data on each iteration, we first calculated average sending time of request, as sending time of each request is very close, we have a reasonable estimate
   - then we calculated average recieving time of responses we get for that burst. Then we apply the **EWMA** formula :

     $$new\_RTT = \alpha * SampleRTT + (1 - \alpha) * Prev\_RTT$$

- we took α to be 0.2 for best results.

In practice the best results we get was from Experiment 1 rather than experiment 2, since our formula is based on extensive experimentation hence experiment gives better estimate than Experiment 2

§3 Code overview

§3.1 Implementation

1. **Initialization**:
   - The server's address and various constants are set.
   - Helper functions for file writing, data extraction, and size parsing are defined.
   - For AIMD without EWMA the **RTT** is estimated in the initial stages first and then that RTT is kept as the waiting time between sending the bursts.
   - For AIMD with EWMA the **RTT** is estimated using the formula:-
   $$new\_RTT = \alpha * SampleRTT + (1 - \alpha) * Prev\_RTT$$
2. **Receiving and Sending**:
   - Two functions are created to handle **AIMD** communication with the server: `receive()` and `send(k)`.
   - `recievie()` listens for data from the server, extracts the payload, and stores it in a dictionary indexed by the data's offset. It also keeps track of received data using a list. it works for till all requests in a congestion window are done or the timeout set for that call is utilised (whichever is earlier).
   - `send(k)` sends k requests to the server based on a list of offsets and sizes, it sends requests in a burst size = k (where k acts the same as **cwnd**). Send used the function `findI()` which acts as iterator and gives the next unreceived offset in **round robin** fashion.
3. **Estimating RTT**
   - For constant window waiting size AIMD algorithm the RTT is estimated by taking the median time taken for first few replies received and 1.2 times the mean.
   - For EWMA implementation , we choose α to be 0.2 as per various experiments and it shows best results in our case.
4. **Main Function**:
   - The `run()` function executes the program.
   - In a while loop which runs until all the offsets are not fetched, we are executing `send(cwnd)` and `receive()` sequentially and at the end depending upon the number of bursts sent and received the cwnd changes.
   - After the data is fetched, the `run()` function reconstructs the received data and calculates an MD5 hash of it.
   - It then sends the **MD5 hash** back to the server for verification and then prints the output in the terminal.

5. **Execution**:
    - The program executes the `main` function if it is run as the main script.

Below is the picture of code segment to show the code.

col334-a3p3 > mani.py > ...

```
    global ind
    global d
    j = 0
    while (j < len(arr) and len(d) < len(arr)):
        x = (ind + j) % len(arr)
        if (arr[x][2] == 1):
            ind = x + 1
            return x
        j += 1

## send a burst of k size
def send(k):...



def receive():...

def writeToFile(data, file):...

def run():
    global cwnd
    global d
    global increase
    global rtt
    start = time.time()
    init = time.time()
    while (len(d) < len(arr)):
        sent = send(cwnd)
        received = receive()
        if (received < sent):
            cwnd = max(1, cwnd // 2)
        else:
            cwnd += increase
        time.sleep(rtt)
    ans = ""
    for i in range(len(arr)):
        ans += d[i]
    writeToFile(ans, 'output_thread_amid.txt')
    md5_hash = hashlib.md5()
```

This is the implementation of run() function which have the sender and receiver functions

§4 Data and Results

1. **For code we adopted i.e Experiment 1**
   - **Parameters**
     a) initial cnwd :- 3
   - **Results**
     a) Server and port:- 10.17.51.115 , 9802
     b) Requested Packets:- 2612
     c) Received Packets:- 1664
     d) Missed Packets:- 948
     e) Time taken:- 21.6873 s
     f) Hash generated:- 39d9ad921281be500ecdf3ea60896b87
     g) Duplicate packets:- 9
     h) Penalties:- 40
     i) Squished:- NO
2. **For Experiment 2**
   - **Parameters**
     a) α :- 0.2
   - **Results**
     a) Server and port:-10.17.51.115 , 9802
     b) Requested Packets:- 2806
     c) Received Packets:- 2281
     d) Missed Packets:- 525
     e) Time taken:- 27.08361 s
     f) Hash generated:- 39d9ad921281be500ecdf3ea60896b87
     g) Duplicate packets:- 13
     h) Penalties:-43
     i) Squished:- NO

§5 Improvement in Time

Compared to previous part 1 there is significant time improvement as shown :-

Without AIMD	With our implementation
86.883 sec	21.6873

§6 Analysis

§6.1 Sequence-number trace

Here is the graph for sequence-number trace also orange is for receiving the request and blue is for sending the request. we can see how our implementation send and receive requests.

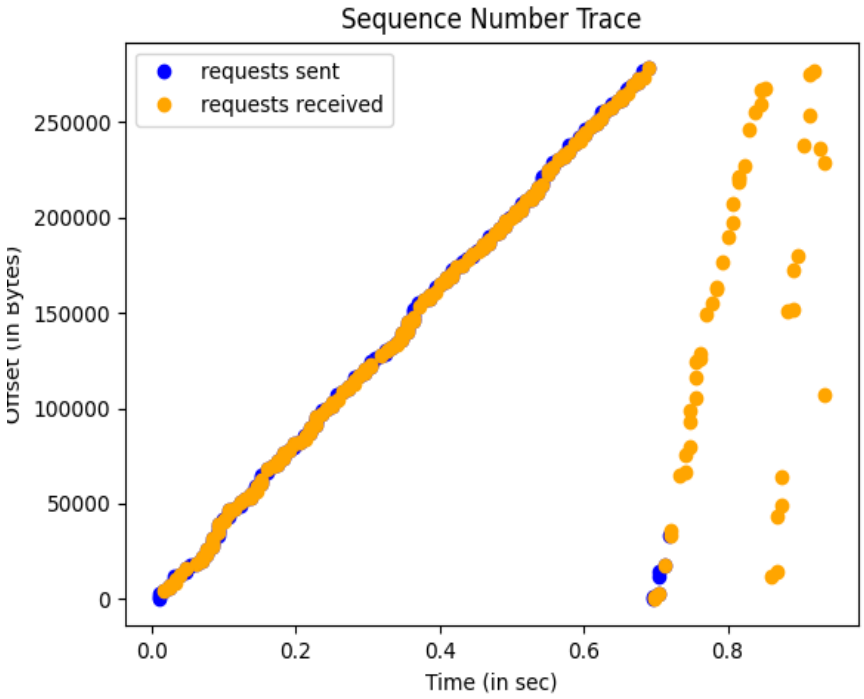

Fig 1: This is the graph for complete time, here vayu server is 10.17.7.134.

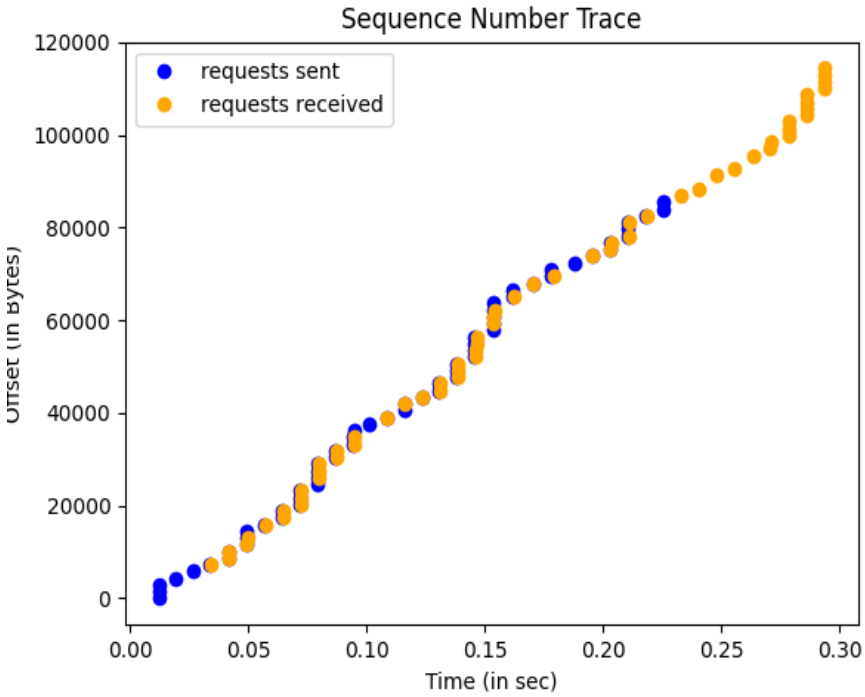

Fig 2: This is zoomed out version of above graph.

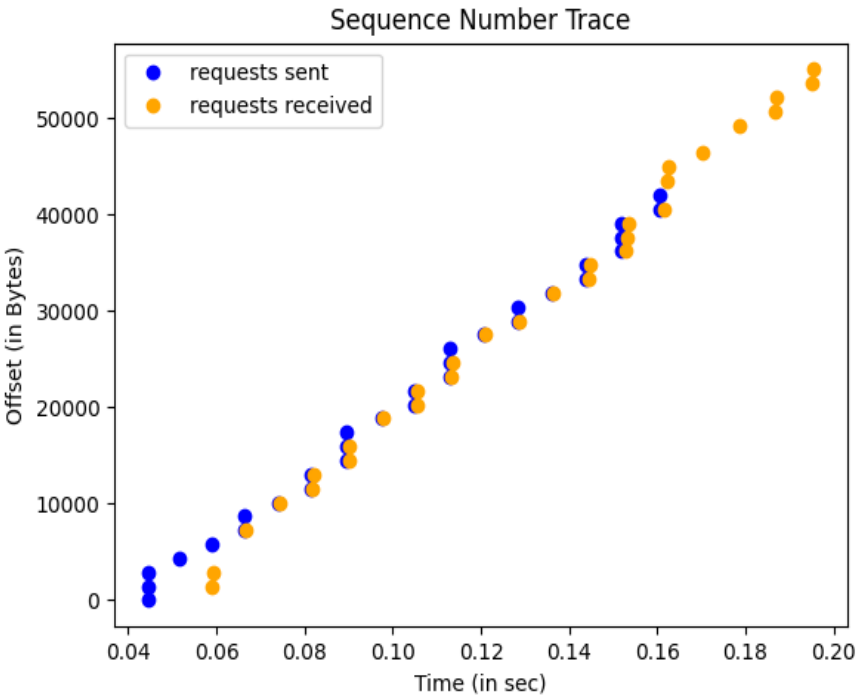

Fig 3: This is hyper zoomed out version of above graph

§6.2 Congestion Window vs Time

This section is to visualise how the burst size vary with time , on part 3 most importnat parameter is burst size which we control. In these graphs, **saw-tooth** , we show how burst size vary over time

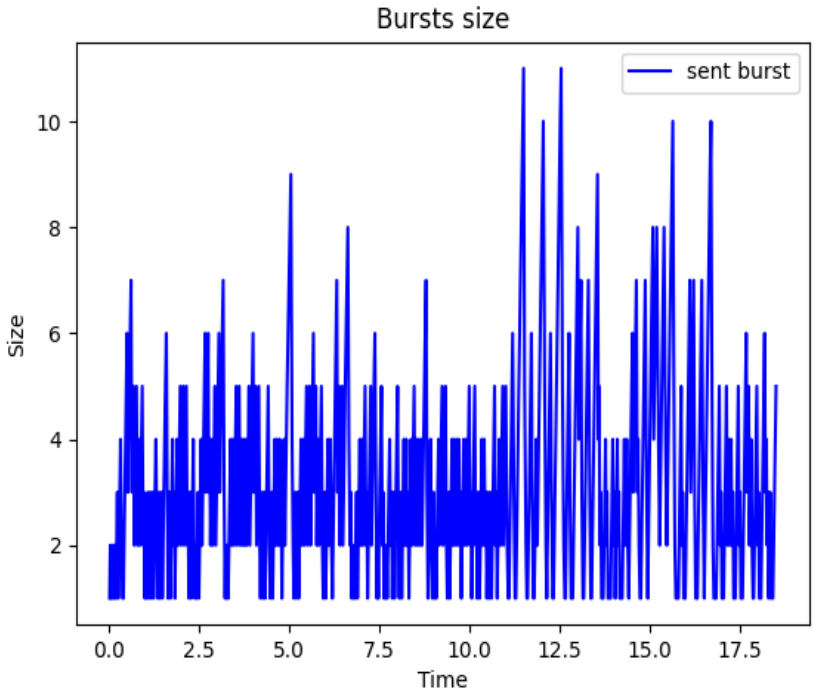

Fig 4: This is burst size/congestion window vs time for full time

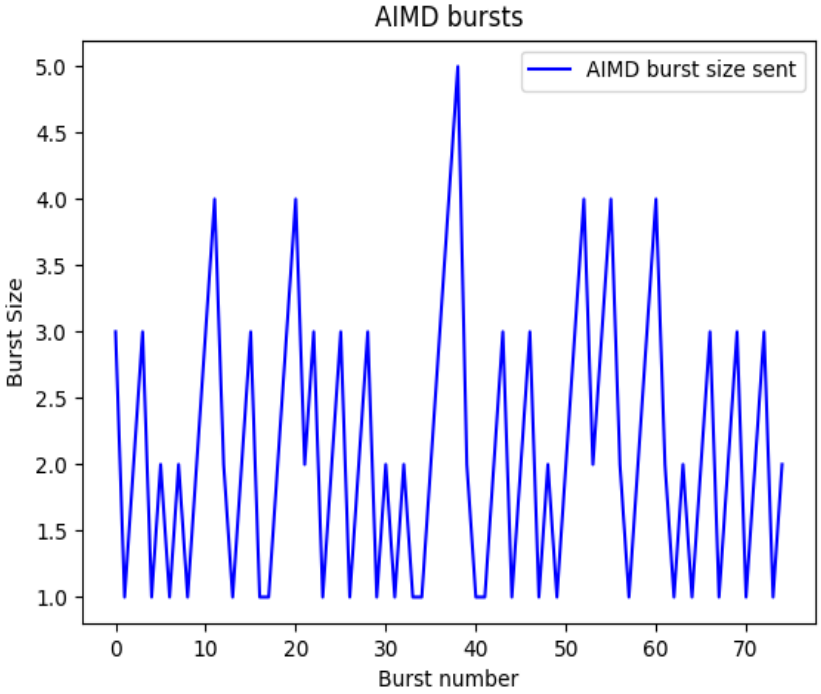

Fig 5: this is zoomed out version of above graph

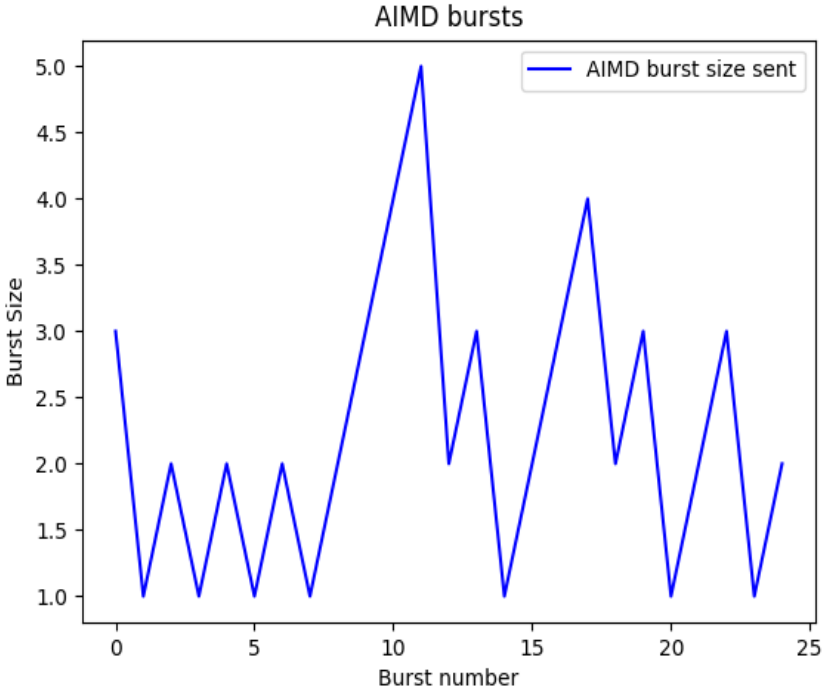

Fig 6: This is more zoomed out version of above graph.

§6.3 RTT vs Time

This section shows the variation between RTT and time. we show two lines in this graph, one is for sampleRTT at each burst and another is net RTT given . we can observe that even though sampleRTT can be very high sometimes net RTT does not show that much spiking due to αbeing 0.2

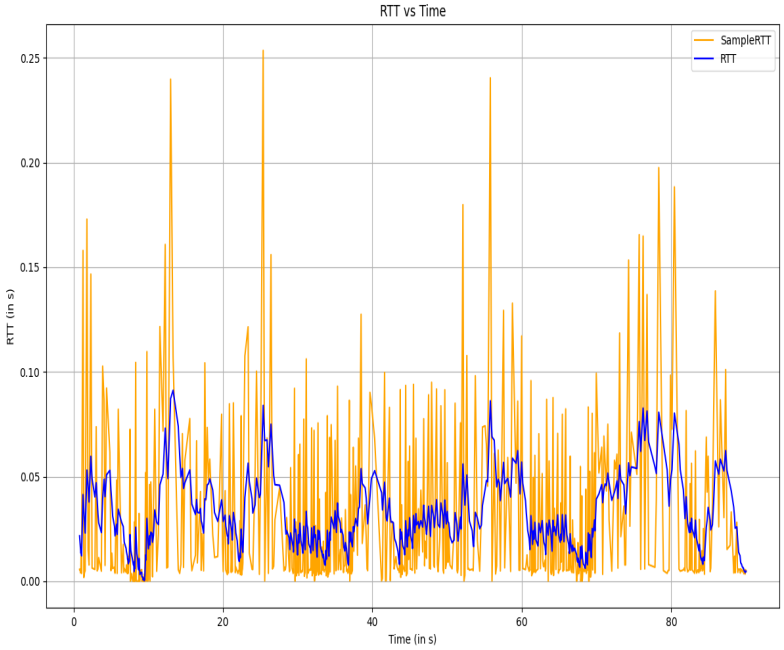

Fig 7: This is graph of RTT vs time.

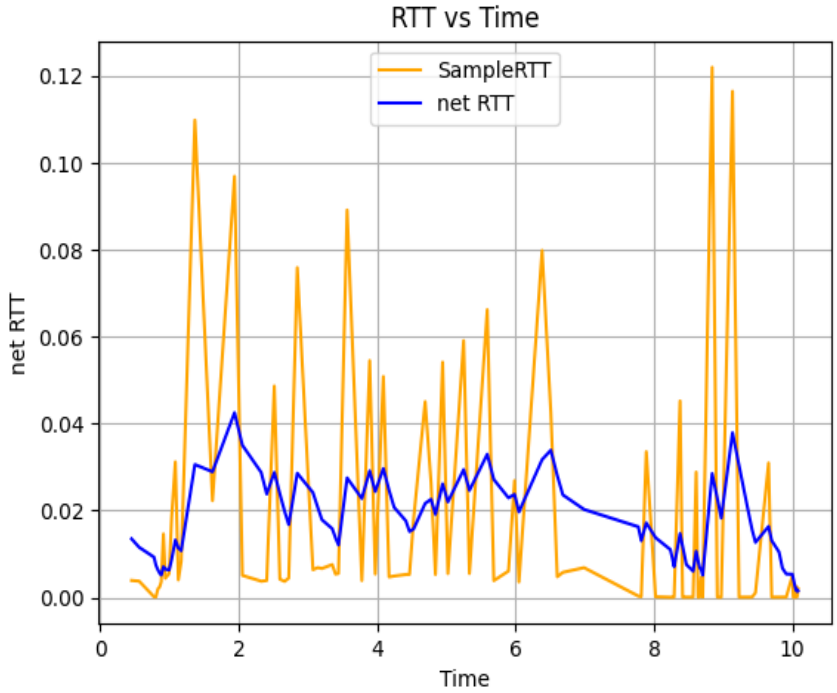

Fig 8: This is zoomed out version RTT vs time.

§6.4 Squish vs Time

This section is to visualise how the squishing vary with time. This is to show how many squish we get with our code in the time. here our implementation we get 0 squishes. Hence our code is working as expected.

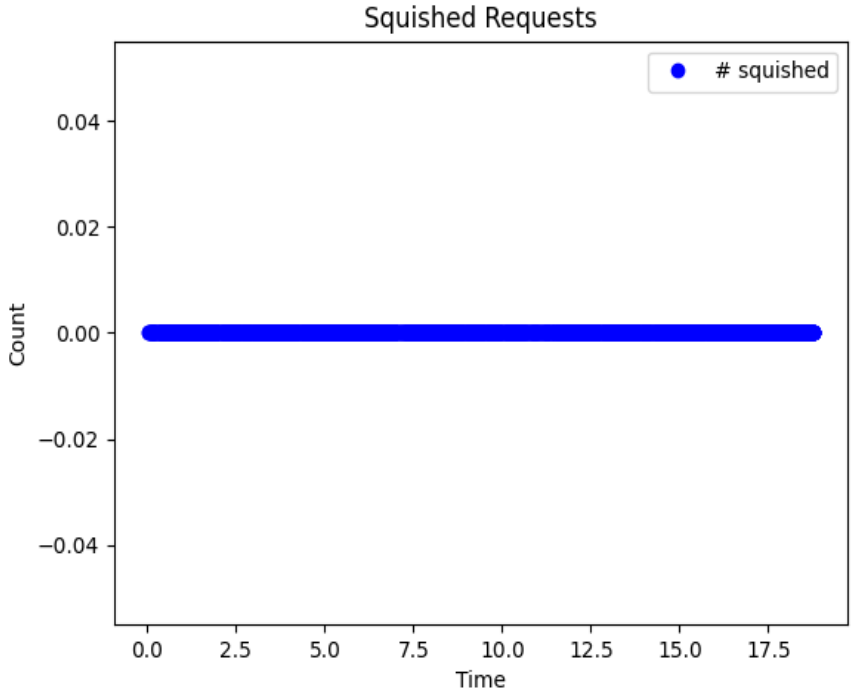

Fig 9: This is number of squishes vs time for full time

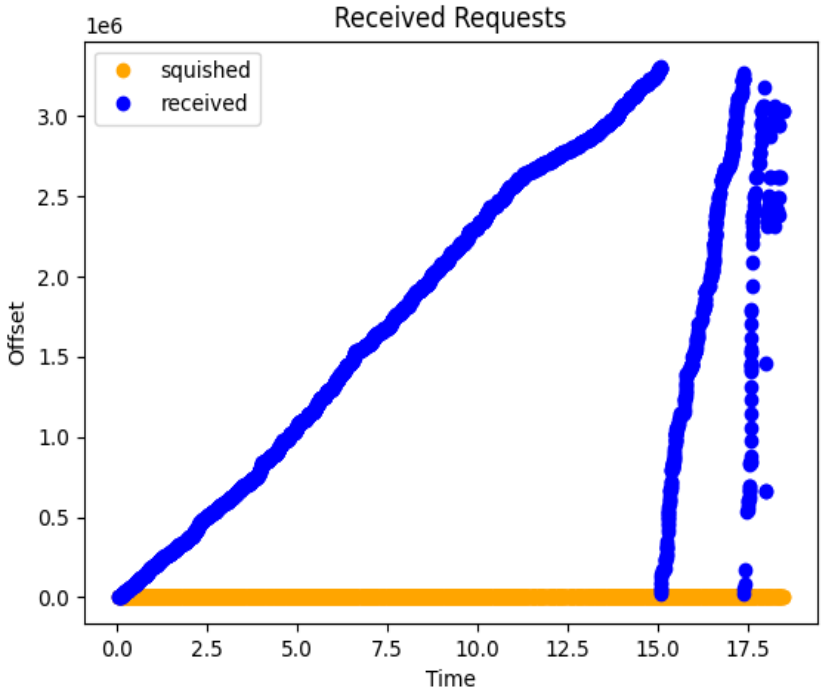

Fig 10: This is to visualise how squish vary with recieved request and time(in seconds).

§6.5 Efficiency vs Time

This section shows the relationship between missed requests and fraction of data fetched versus time.

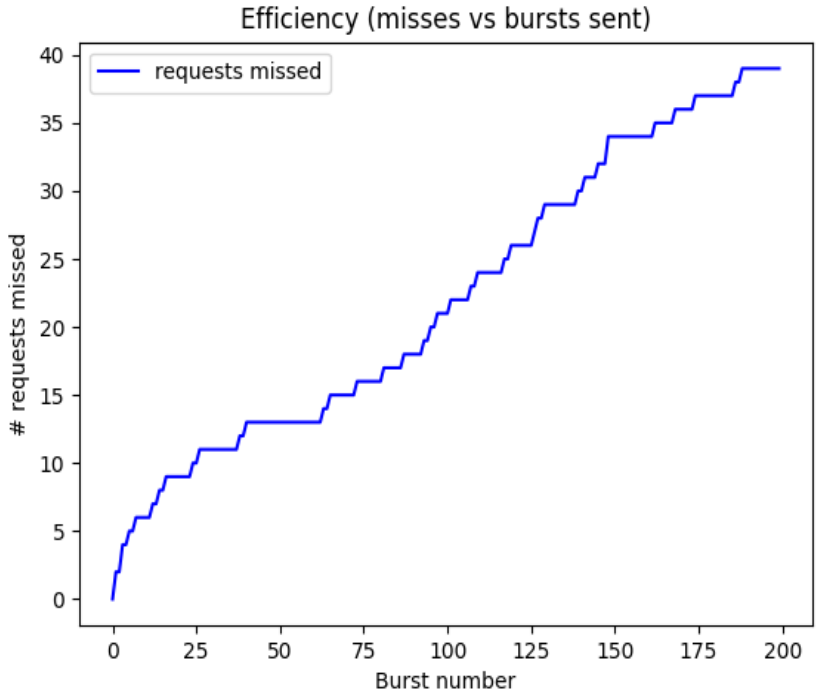

Fig 11: This is to visualise the percentage of data received w.r.t time.

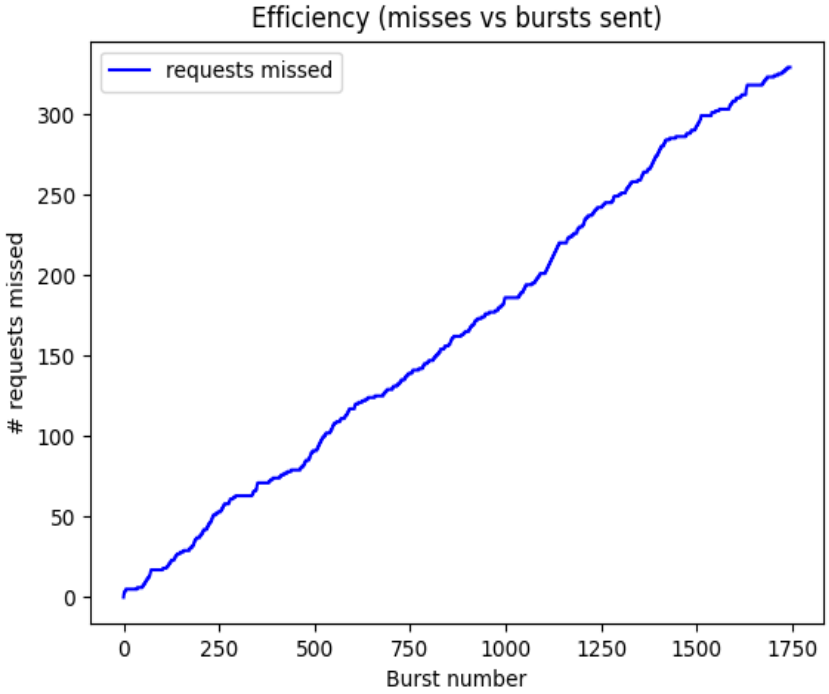

Fig 12: This is to visualise how many misses have been there on with the requests sent

§6.6 Constant size burst

This section shows that sending constant size burst will not give better implementation in variable server rather it gives worse results. As we can see in the graphs more the burst size more the client is squished most of the time and low burst size shows no improvement in time.

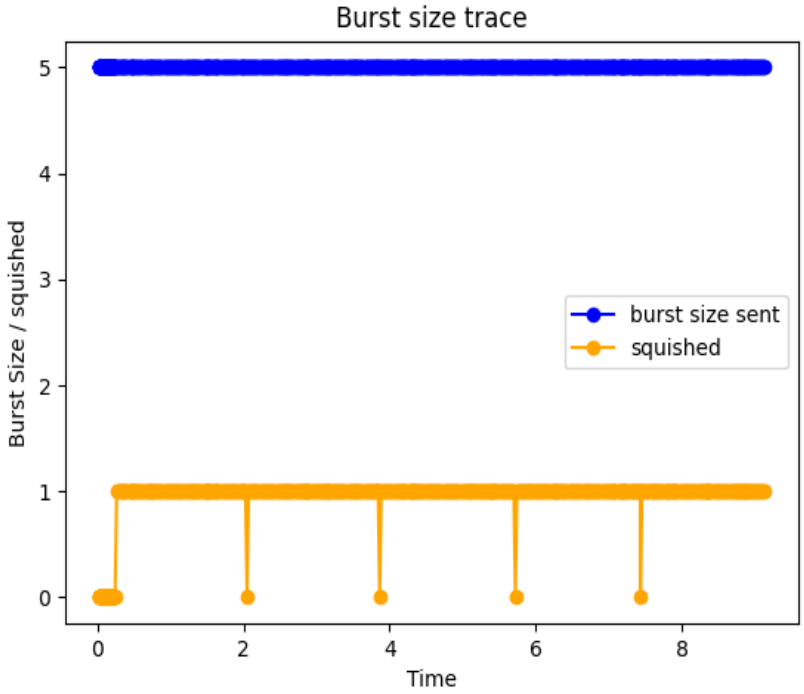

Fig 13: for burst size 5 we observe the server is getting squished earlier

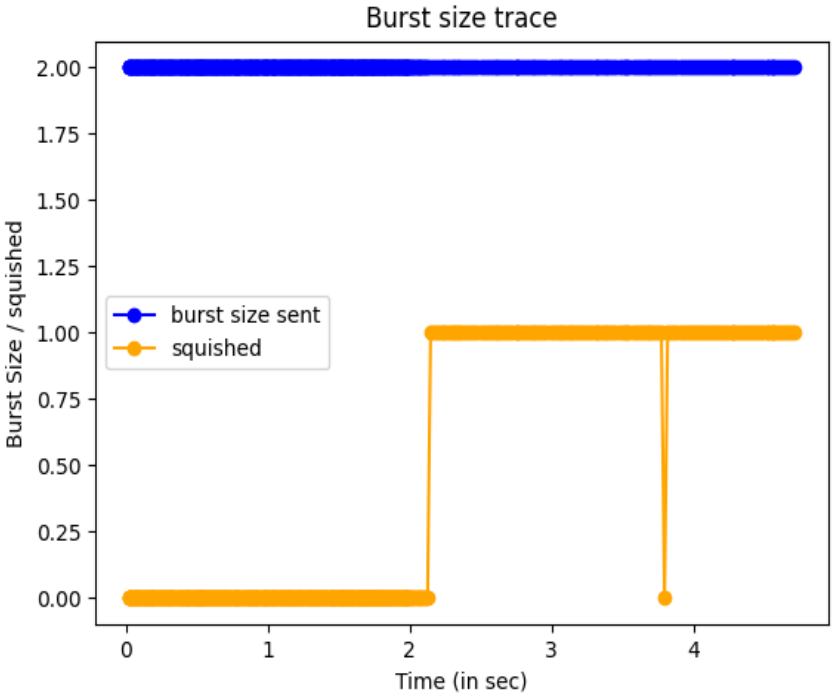

Fig 14: With bursts of size 2 we find the server getting squished lately but it is also getting squished

§6.7 Bytes vs Time

Here is bytes vs time analysis, we observed how time increases as we increase the bytes of data. We observed the following curve, where we observed that time is directly proportional to bytes taken; they vary linearly with each other. Also since this time we have variable rate server the lines are not perfectly straight but have a bit of variation.

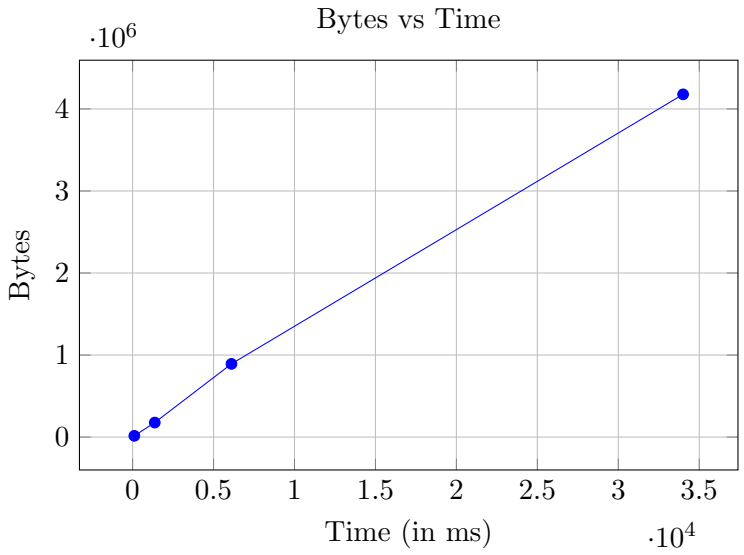

Fig 15: Bytes vs Time

§6.8 Lines vs Time

Here is the Lines vs Time analysis, where we observed how time increases with the number of lines. We can see that time and the number of lines are positively correlated but not directly related. This followed the expected outcome.

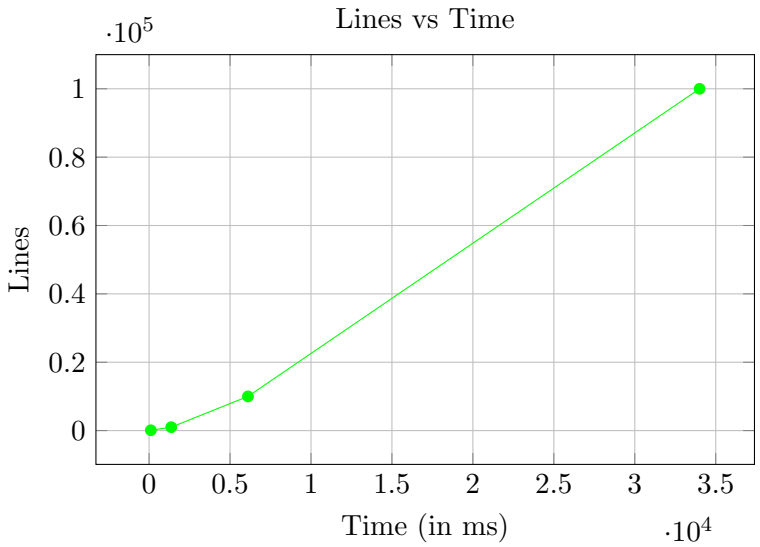

Fig 16: Lines vs Time

§7 Exception handling

- in case of packet drop, we send the request for it again so we can recieve it in next iteration using **round robin** method.
- in case of skipped request, it is handled in the same manner as above we send the request for it again till we get the appropriate response
- Also we handled exception for submitting the text, in case that packet itself for submitting the data is dropped, then we run a **WHILE** loop till we get the packet with the acknowledgement that data is submitted