COL334 Assignment 2

Dhruv Ahlawat 2021CS10556
Tejas Anand 2021CS50595

October 2023

1 Introduction

In this assignment, we attempt to build a reliable data transfer protocol over servers using UDP. The servers will implement a leaky bucket, that maintains tokens for every user that sends requests to it. Each time a request is sent the reply is sent only if that user has some non-zero tokens. The tokens replenish with some constant rate R but that R is often slow enough to prohibit clients from sending very rapid requests which tend to congest the network. As a result when the server receives a request when the client has no-tokens, it squishes them by reducing the rate of generation of their tokens. **Note: details about implementation in Checkpoint 2 is mentioned at after checkpoint 1 and labelled as checkpoint 2**

2 Checkpoint 1: : Receiving Data Reliably

- As we know, UDP is a fast and unreliable protocol. That means that, there is no gaurantee the responses to the requests we make to the server are actually going to make their way back to the client.
- So, the client has to keep track of what it has received so-far and what further needs to be asked for.

3 Approach 1: Reliable Sequential Data Transfer[Checkpoint 1]

- In Sequential Data Transfer, we obtain the max-size of the response and the chunk-size via the **SendSize** request.
- Then we sequentially iterate over the offset numbers, we send the request for a particular offset numbers with the request size equal to the chunk-size, and then wait for the response for that particular offset.
- Since the server may decide to arbitrarily drop the response, we set a parameter time-out after which we request data for the same offset again.
- After receiving the response, we sleep for sleep-time before sending the next-request.
- This approach ensures reliability as any of the packets that is not received is waited for until it is received
- The time taken by this implementation is highly dependent on the sleep-time.

4 Approach 2: Burst Data Transfer[Checkpoint 1]

- In Burst Data Transfer, we obtain the max-size of the response and the chunk-size via the **SendSize** request.
- We maintain a set of offsets for which we have not obtained the data.
- This implementation works in multiple-passes instead of a single pass.
- In one pass, we iterate over the set of unobtained offsets and start requesting for data in the sizes of chunks.
- After requesting for a particular chunk, we wait for the responses for each of the offsets in that chunk.
- But, in this approach we don't keep waiting for the response of a particular offset in the chunk which we don't get. Instead we would later get the data for that offset in a later pass.
- The time taken by this implementation is highly dependent on the sleep-time and chunk-size.

5 Approach 3 : Threaded Burst Data Transfer [Checkpoint 2]

- This is similar to the above approach except that now we are sending and receiving in separate threads
- We have two threads, one thread for sending data and one thread for receiving data. These threads share a common resource that is the set of offsets which we have not received.
- The sending thread is running in a loop which stops as soon as the size of the set becomes zero.
- The receiving thread reduces the size of the set when it receives the data of a particular offset.
- We send the requests both of sending and receiving in bursts.
- This implementation does not keep track of the data packets that we lose, and hence is a very basic implementation useful only when we know the ideal rate of data transfer and is constant.

5.1 Graphs for burst data

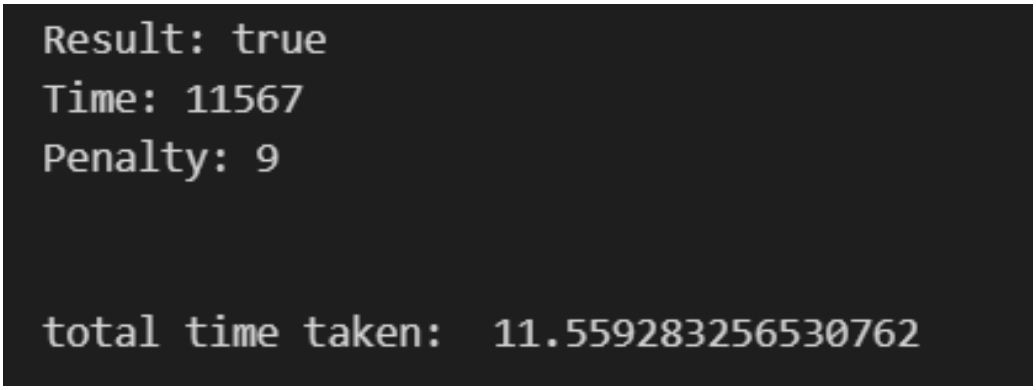


Figure 1: result on burst threaded data

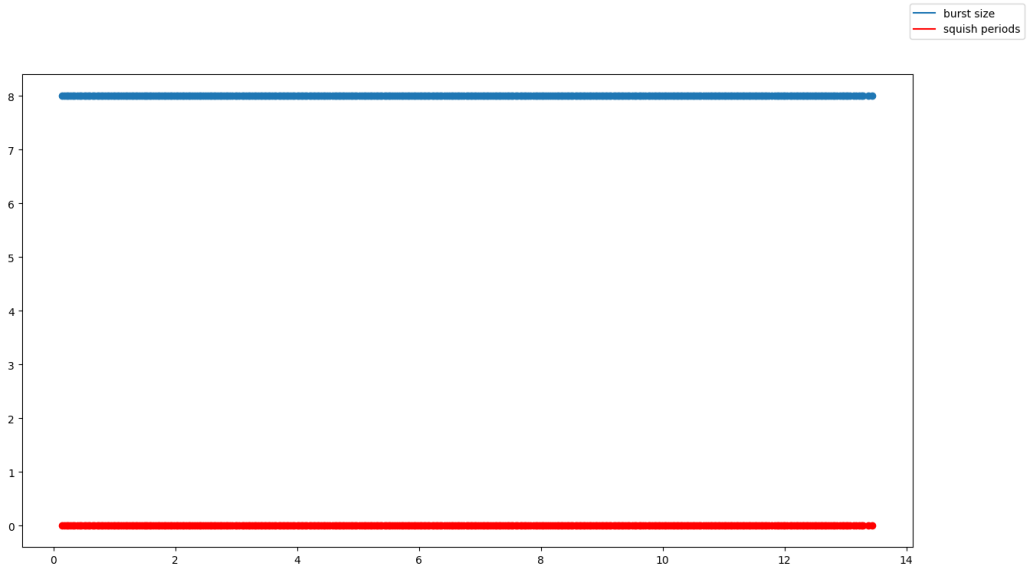


Figure 2: squish periods and burst size graph

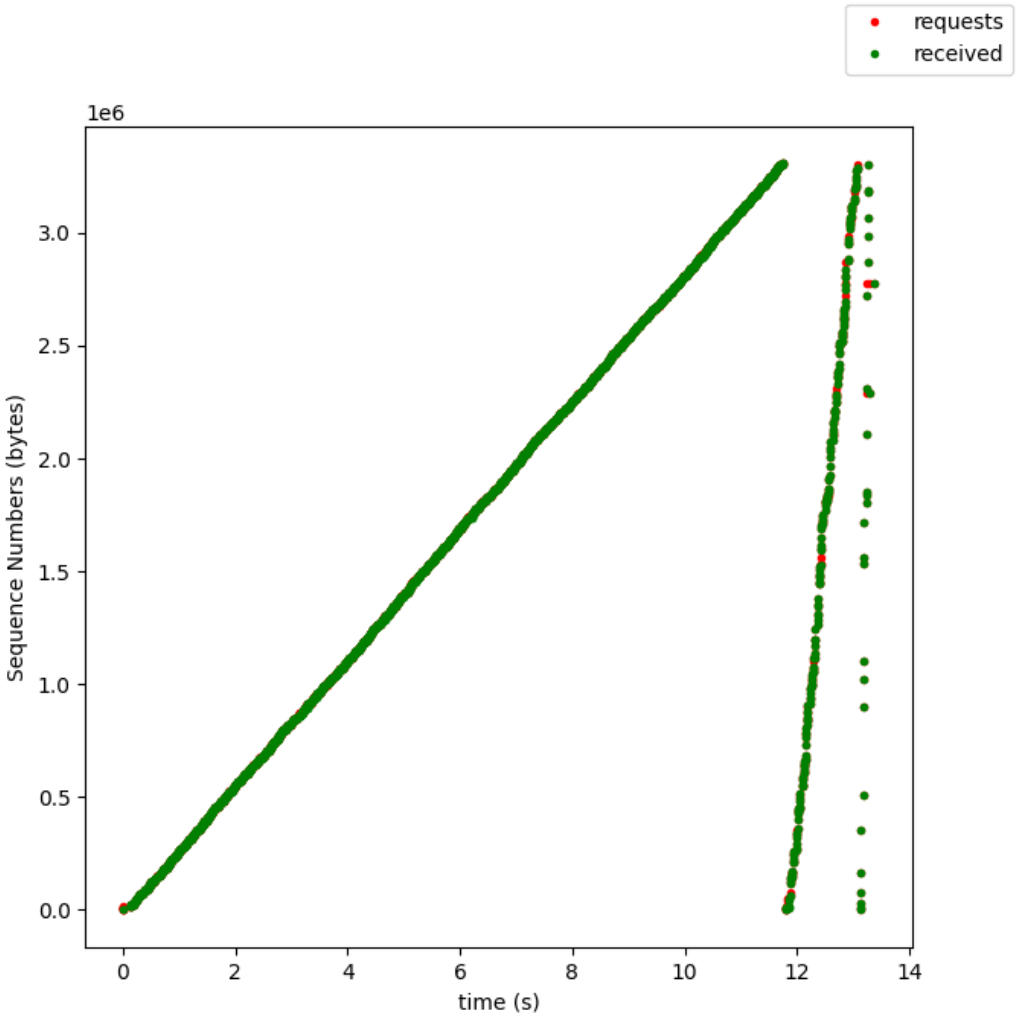


Figure 3: Sequence numbers vs time

6 Approach 4: AIMD [Checkpoint 2]

- Knowing that the leaky bucket parameters are constant for checkpoint 2, we implemented a **minimum and maximum limit** for our burst rate parameters.
- The **timeout** is calibrated using a multiple of the RTT calculated using exponentially weighted moving average.
- We are able to get times of upto 13 seconds using this method on vayu server, with penalty under 20.
- We also experimented with calculating the average rate of dropping of packets, and found that to be 10%. Using this, we can decide when we must decrease our rate of sending if the packet loss exceeds this rate, otherwise the loss can be considered to be from **random dropping instead of due to congestion**.

7 Approach 5 : AIMD with constant timeout

- This is similar to the previous approach except after sending a burst, we wait for the burst to be received for a constant time.
- That constant time is set as a multiple of the RTT (x*RTT).
- We can see from the data , timeout = 2RTT gives us the best result

x	1	2	3	4	5
Time	13.37	12.12	13.59	13.87	13.9

7.1 Graphs for AIMD

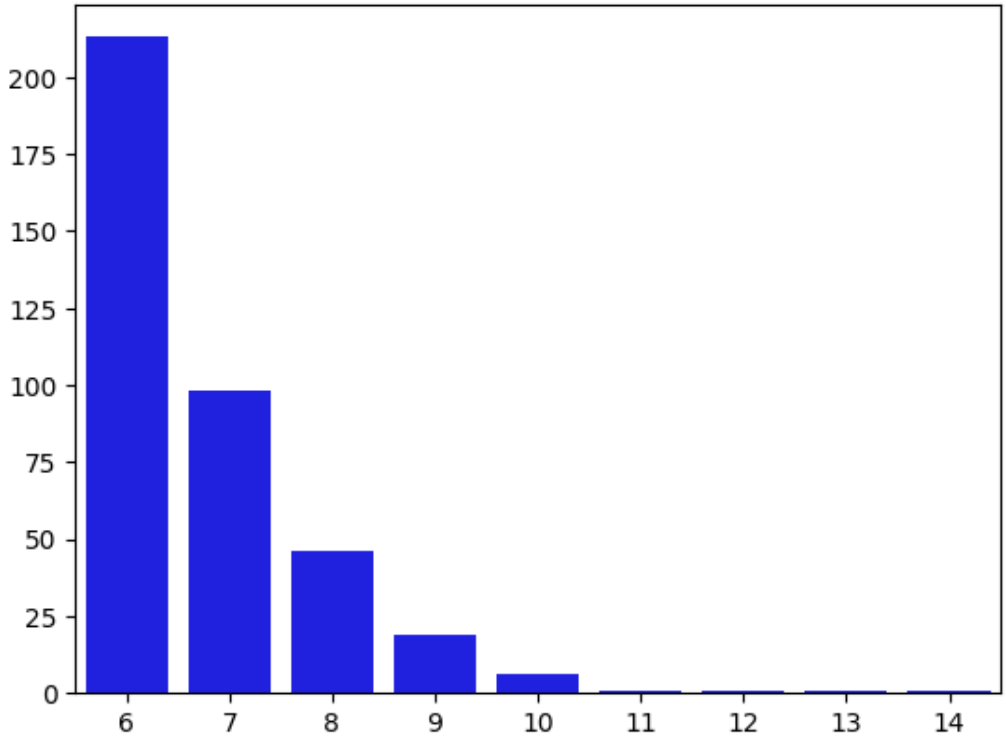


Figure 4: frequency of burst rates (minimum = 6)

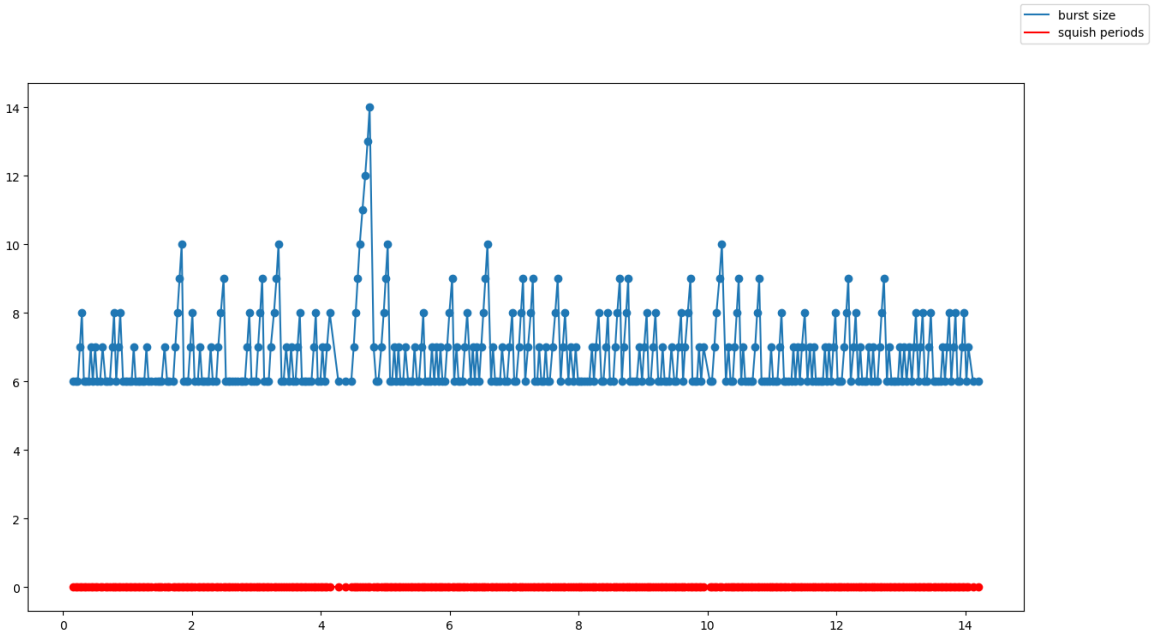


Figure 5: burst rates vs time and squishing vs time

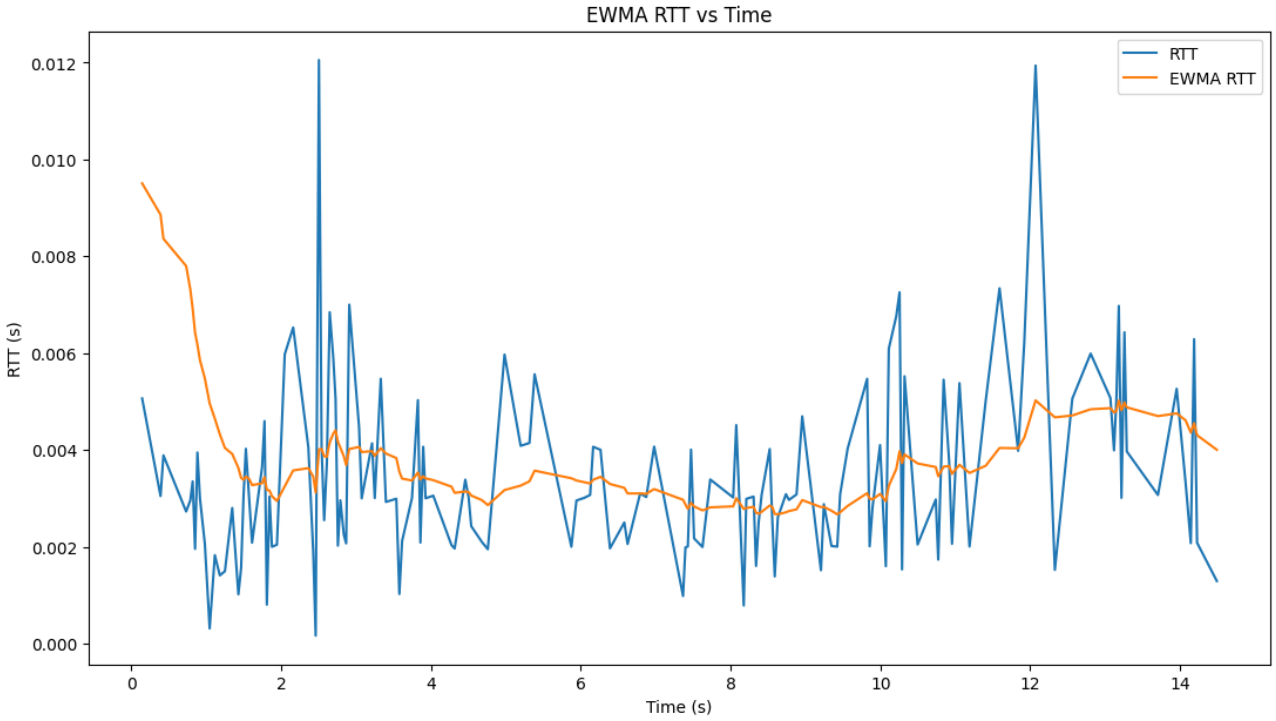
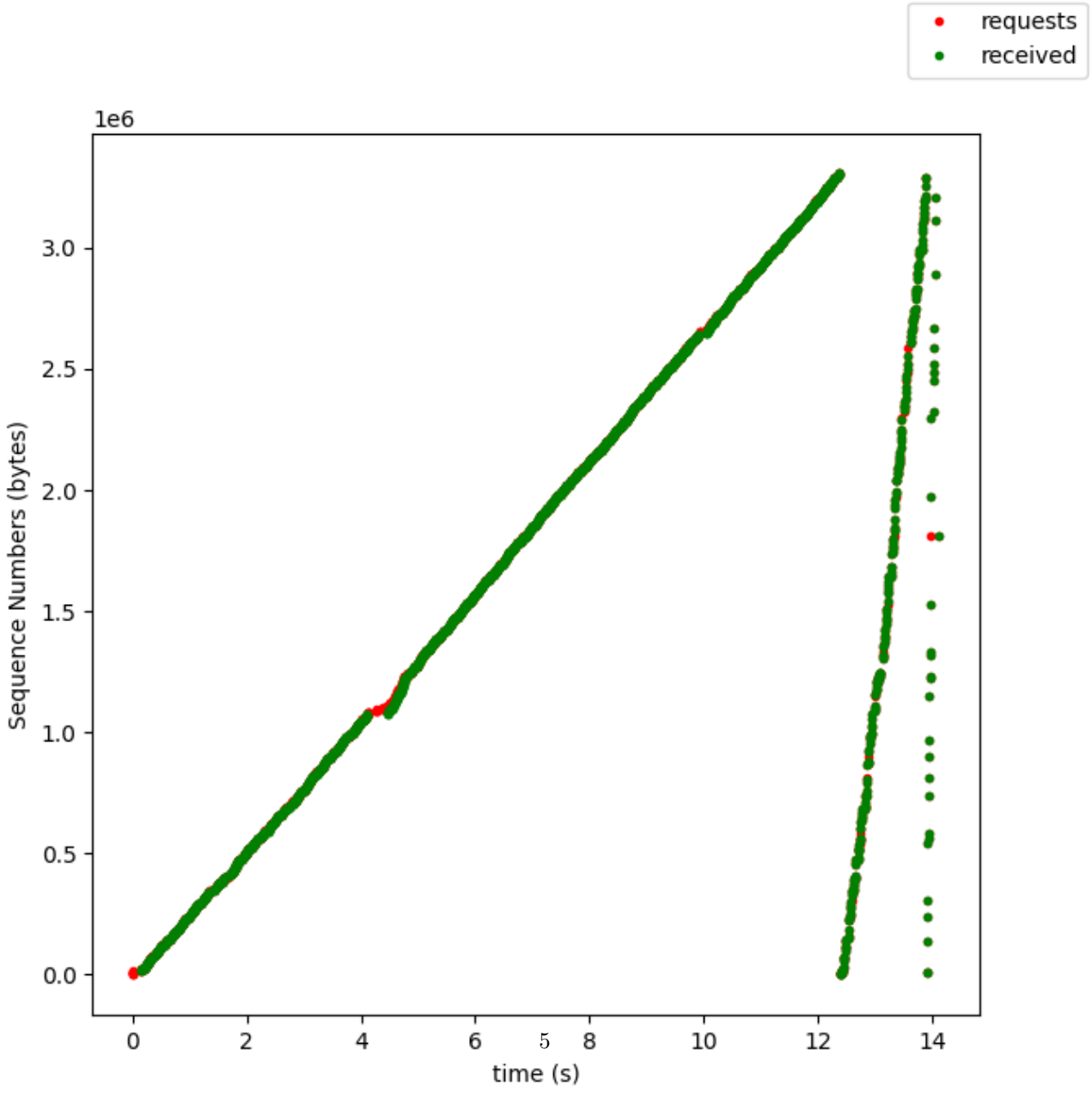


Figure 6: EWMA RTT measurement, weight alpha = 0.2



8 Observations and Graphs

8.1 Sequential Data Transfer

	Sequential Data Transfer	
Sleep-Time (ms)	Time Taken (s)	Penalty
100	71	0
50	40	0
10	14	0
5	9.04	0
2	11.25	35

- Here we can see that as the sleep time decreases, the time taken to get the file decreases linearly,
- But after a certain threshold if we try to decrease the sleep-time, the requests become too fast and the time taken to get the file infact starts increasing due to the penalty of fast requests.
- We also observe the proportion of unanswered requests increases as we decrease the sleep time.
- We can observe in the zoomed picture we get the response of each request almost simultaneously as sleep-time is high (100 ms)
- When sleep-time is low, many requests get unanswered (2 ms)

8.2 Burst (Constant Size) Data Transfer[Checkpoint 1]

- We observe that the majority of requests get responses in the first pass itself and the entire data takes atmost 3 passes to get downloaded.
- We observe that the Time Taken is inversely related to the burst size.
- We get close to sequential's performance at low sleep-times (5 ms) even at comparitively high sleep times (20 ms). This is due to the burst nature of the requests.
- We can't reduce our sleep-times to the levels of sequential sleep-times (1-10s) because the burst nature of the requests causes penalties and start squishification of the network. There exists a threshold value of the burst size for each sleep time after which squishification starts happening.
- In the zoomed in graphs we can observe that the entire burst doesn't get the response in one pass but a majority of them do.

	Burst Data Transfer		
Sleep-Time (ms)	Burst Size	Time Taken (s)	Penalty
200	7	26	0
200	3	54	0
50	7	11	0
50	3	17	0
20	7	8	1
20	3	10	0

8.3 Threaded Burst Data Transfer[Checkpoint 2]

- Threaded Burst Data Transfer is much faster than normal constant size data transfer.
- On our local servers, this implementation on average gave us 3s of time to submit the entire file successfully

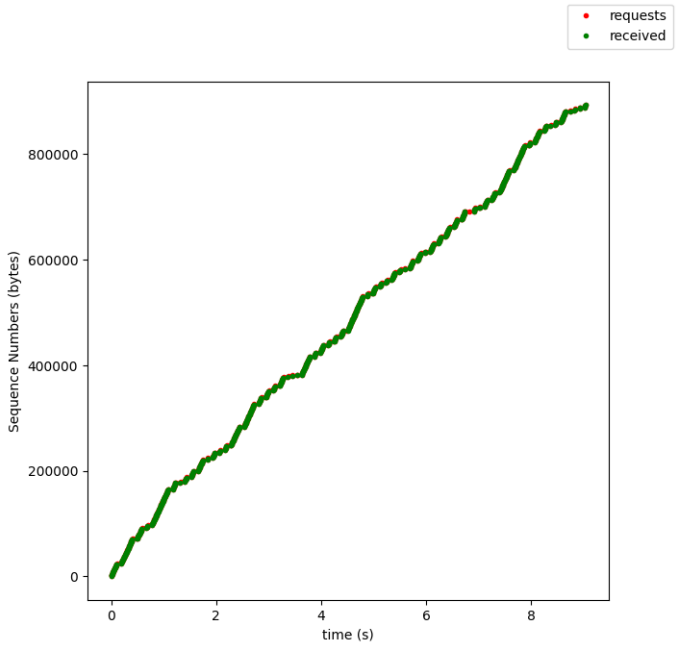


Figure 8: Sequential: Delay = 2 ms

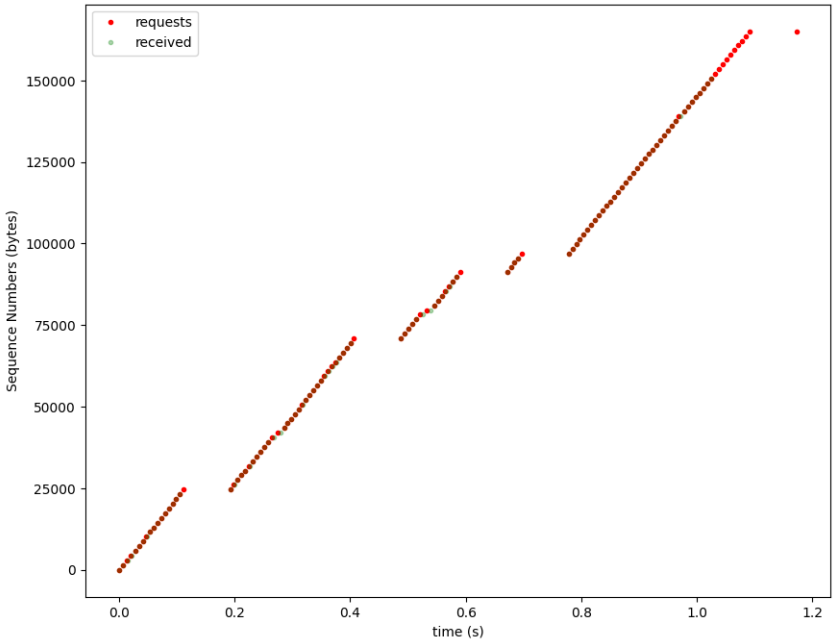


Figure 9: Sequential: Delay = 2 ms

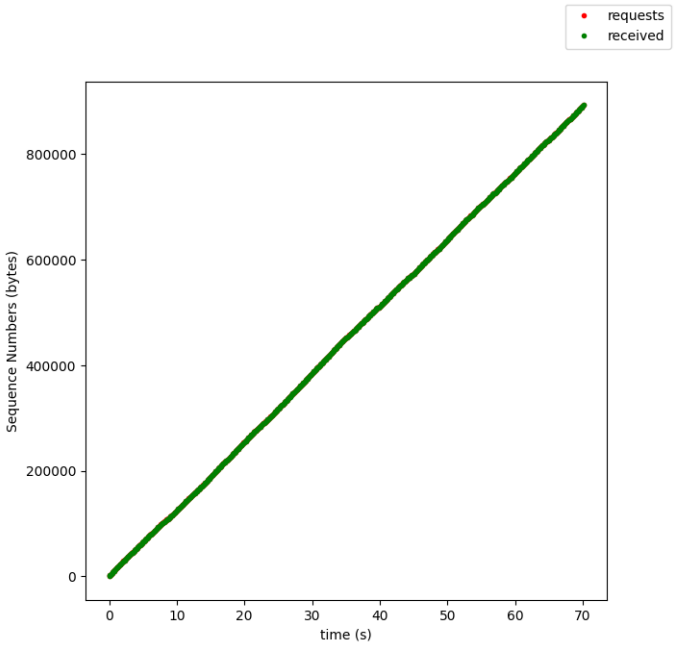


Figure 10: Sequential: Delay = 100 ms

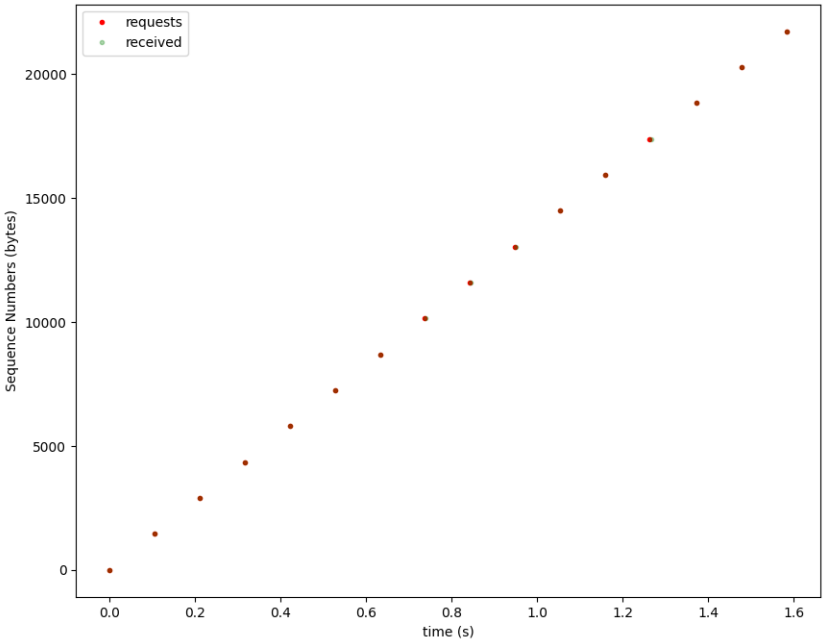


Figure 11: Sequential: Delay = 100 ms

8.4 AIMD with EWMA Weighting[Checkpoint 2]

- We tried many different values of the multiplicative factor and the additive factor.
- The optimal additive factor was 1 and the multiplicative factor came out to be about 1/1.7.
- We aren't getting squished, so our squish period is 0.
- The frequency of burst size is inversely related to the burst size. The most frequent burst size was 6.
- The graph of EWMA with time smoothens out the irregularities present in the RTT.

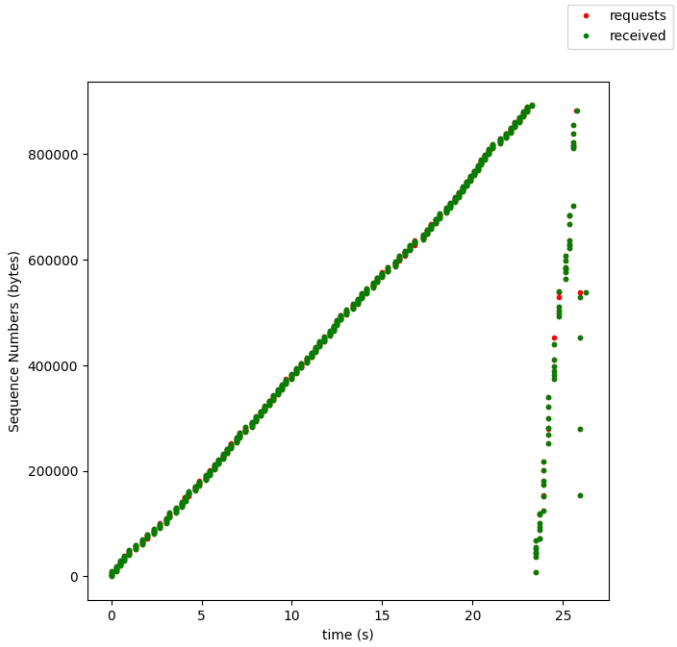


Figure 12: Burst: Delay= 200 ms, Burst-Size=7

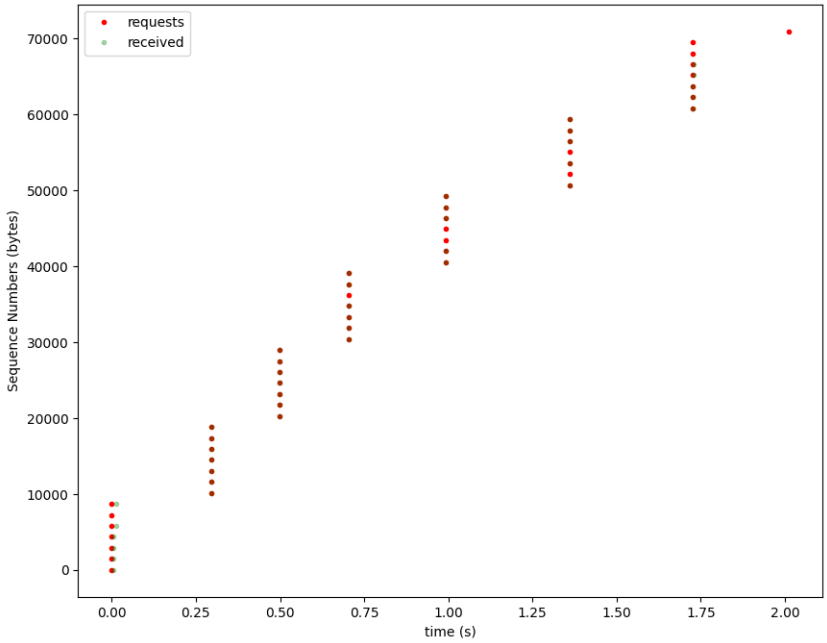


Figure 13: Burst: Delay = 200 ms, Burst-Size=7

8.5 AIMD with EMWA RTT proportional rate [Checkpoint 3]

- We have 2 parameters in this modified AIMD implementation, timeout and the rate (sleep_time).

- The timeout is the time for which we wait for a packet before considering it to be lost on the network.
- Rate is the time between receiving of the last response and the sending of the next request.
- In this variation, we keep the timeout and the sleep_time both proportional to the instantaneous EWMA RTT.
- We play around with the constant factors to figure out the best constant factors.
- We expect this approach to work as the EWMA would be a measure of how congested the network is. So, if the server decides has a variable rate of token generation and decides to drop some packets, then the EWMA would be slightly affected by this and we would slow down (as the sleep time is proportional to EWMA) to reduce the penalty and prevent squishification.
- We would also be changing the burst-rate along with the sleep_time whenever a packet is dropped.
- However, this approach has a drawback, which we would discuss along with the graphs.

8.6 AIMD with EWMA RTT dynamic proportional rate [Checkpoint 3]

- This is similar to the above approach except here we not only depend on the ewma_rtt to slow us down, but we also modify the constant factor in the sleep time.
- This means that in the expression of sleep_time = x*ewma_rtt, we also change x when we notice packets are being dropped.
- The severity of the changes in x are dependent upon the number of packets that are dropped.
- When this number is 1, we mildly increase x, and mildly decrease the burst size.
- When this number is 2, as the probability of 2 packets being dropped simultaneously is low, it is likely that we have emptied our bucket and we should slow down a bit more steadfastly.
- When this number is 3, we are almost sure that we have emptied our bucket. We need to increase x and decrease the burst size drastically.
- When this number is greater than 4, we are in grave danger and we decide to set x to its upper bound and set burst rate to 1, and also additionally wait for some sleep time. (This was not part of the implementation earlier, but we noticed that we never get squished when we do this change which is kind of obvious given how this event is a proxy for being squished)
- In case we get squished, we set our burst rate constant and slow down to a constant sleep time. (This never happens though)
- We played around the parameters of all of the above cases to figure out the best ones.

8.7 Graphs for AIMD [Checkpoint 3]

note that in figure 16, the start is not actually from squished state, but that is because it was initialized that way. The actual sending and receiving happens a bit later and the variables have their proper value at that time.

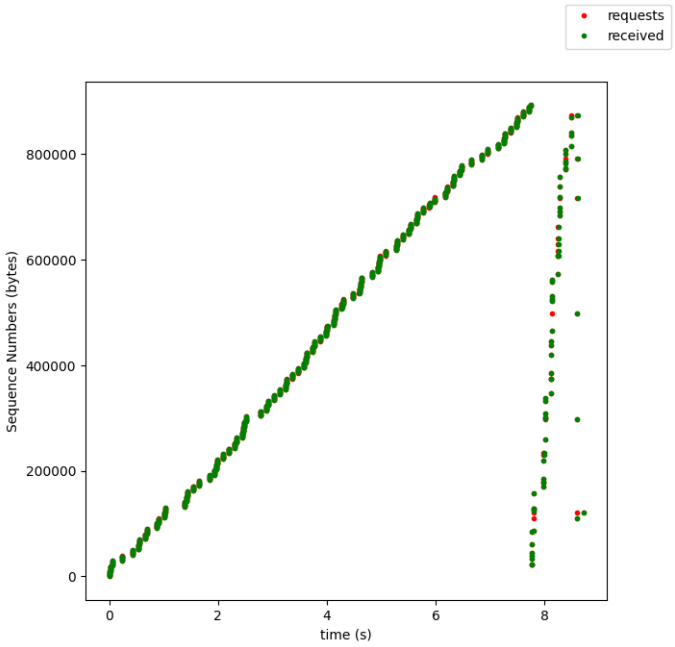


Figure 14: Burst: Delay= 20 ms, Burst-Size=7

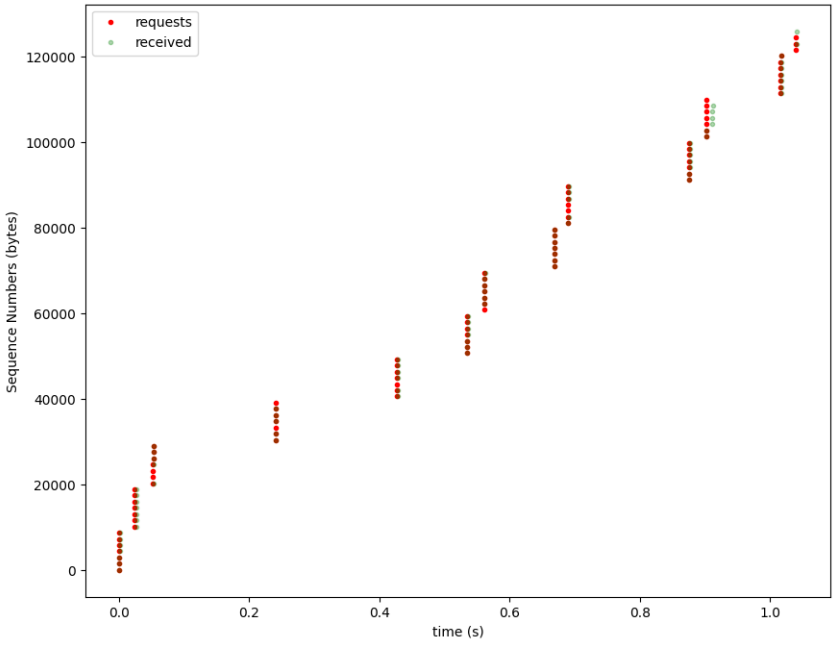


Figure 15: Burst: Delay = 20 ms, Burst-Size=7

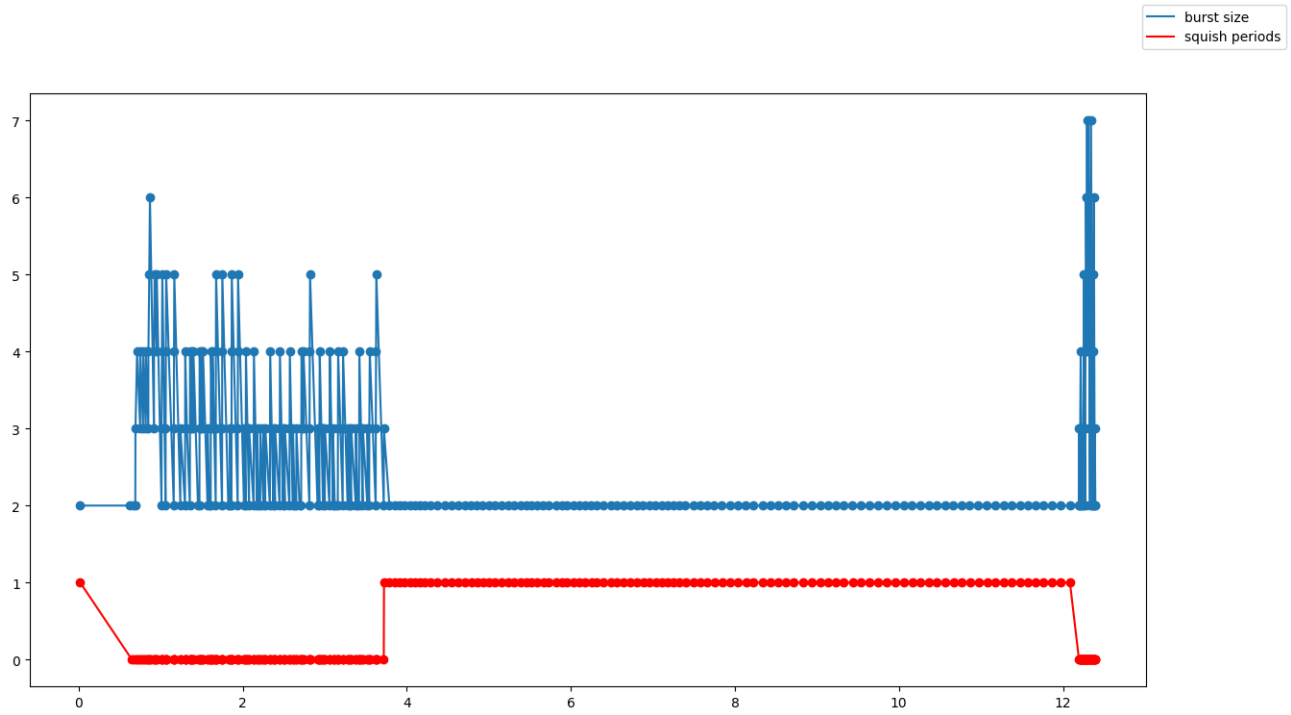


Figure 16: Burst Sizes and Squish Periods vs Time in Constant x

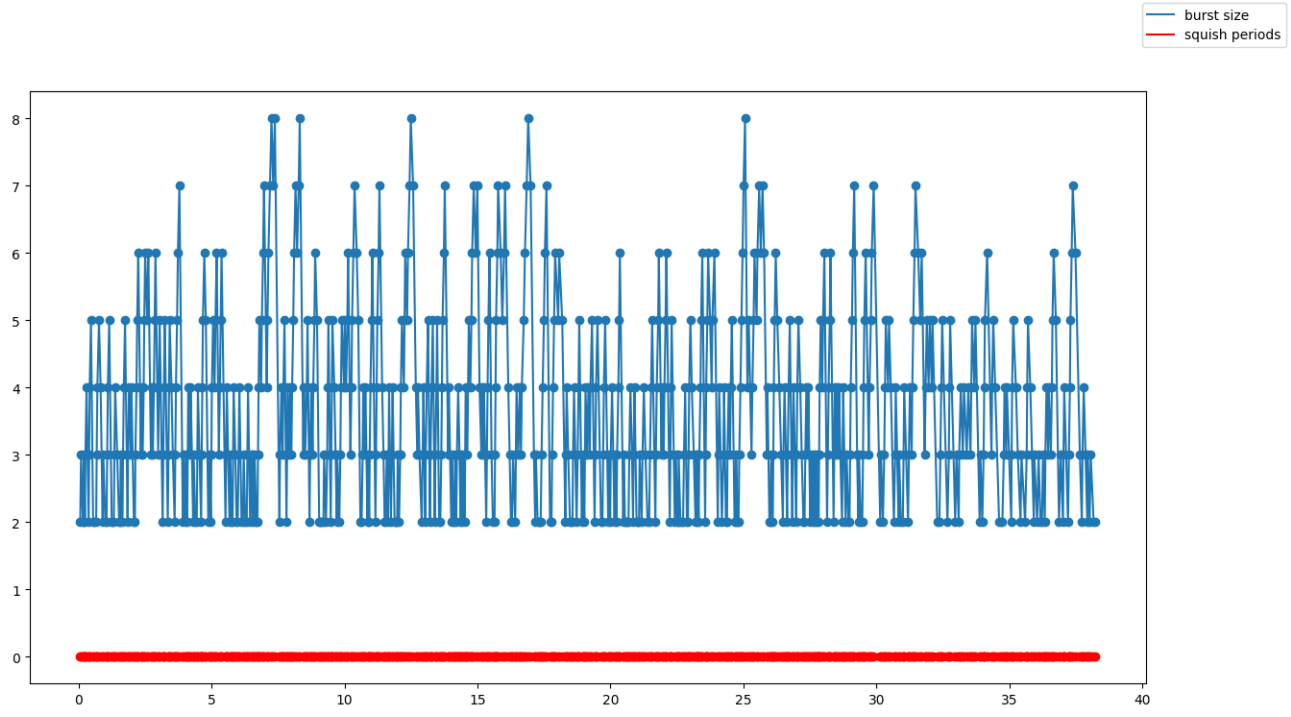


Figure 17: Burst Sizes and Squish Periods Vs Time in Dynamic x

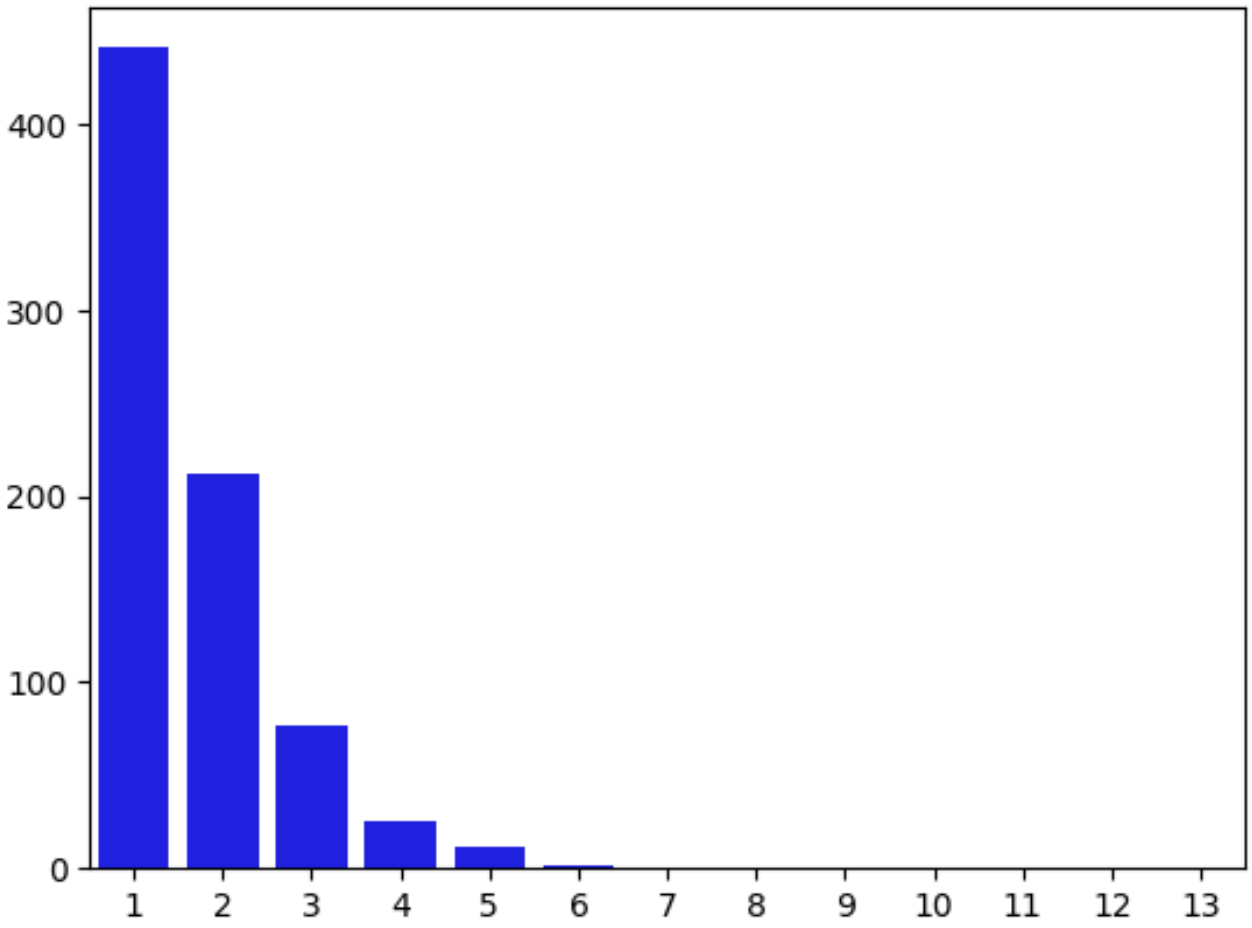


Figure 18: Frequency of dropped packets

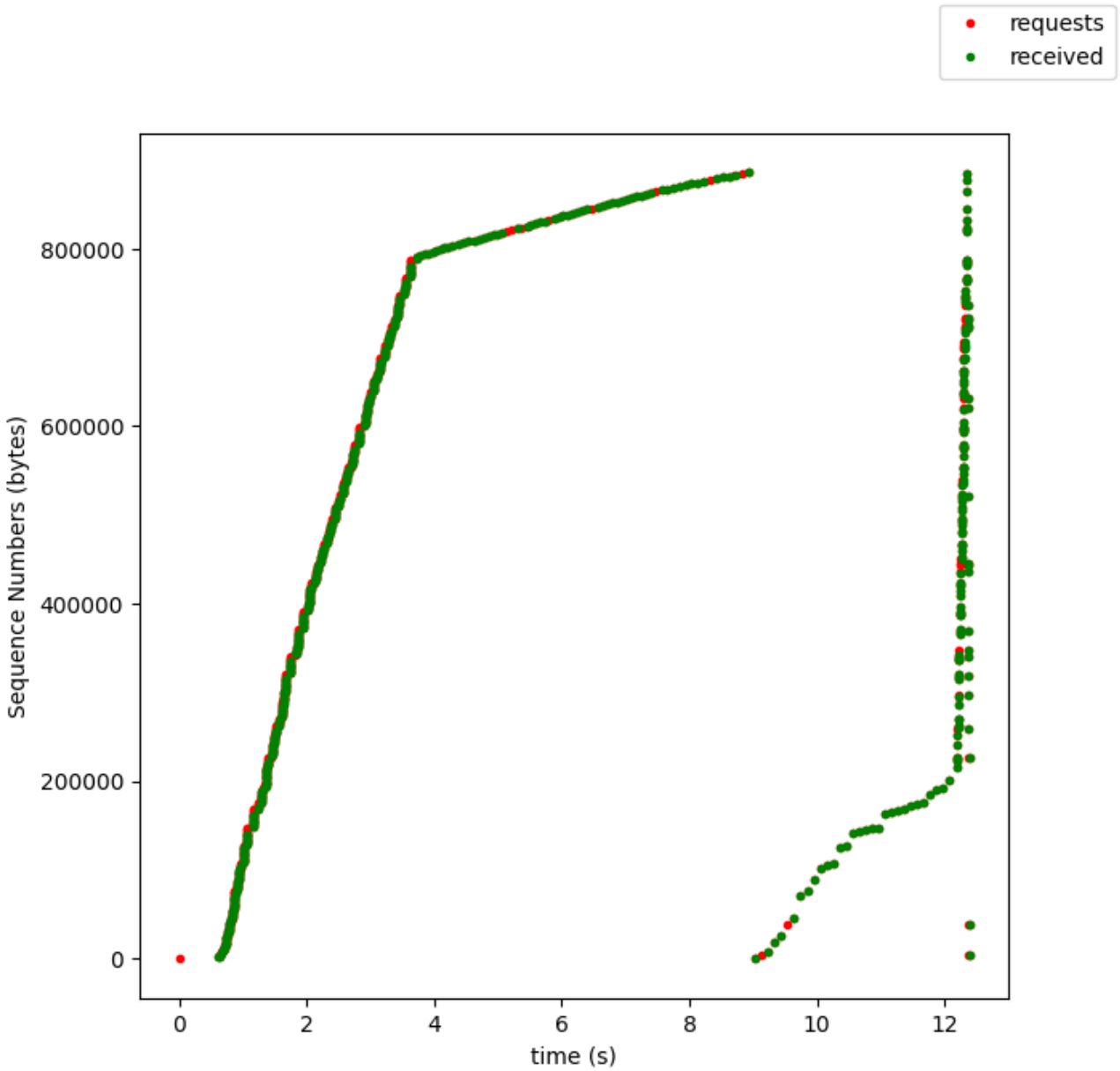


Figure 19: Sequence Numbers in the variable rate server with constant x implementation, squish starting at around 4 seconds. Recovering from squish at 12 seconds

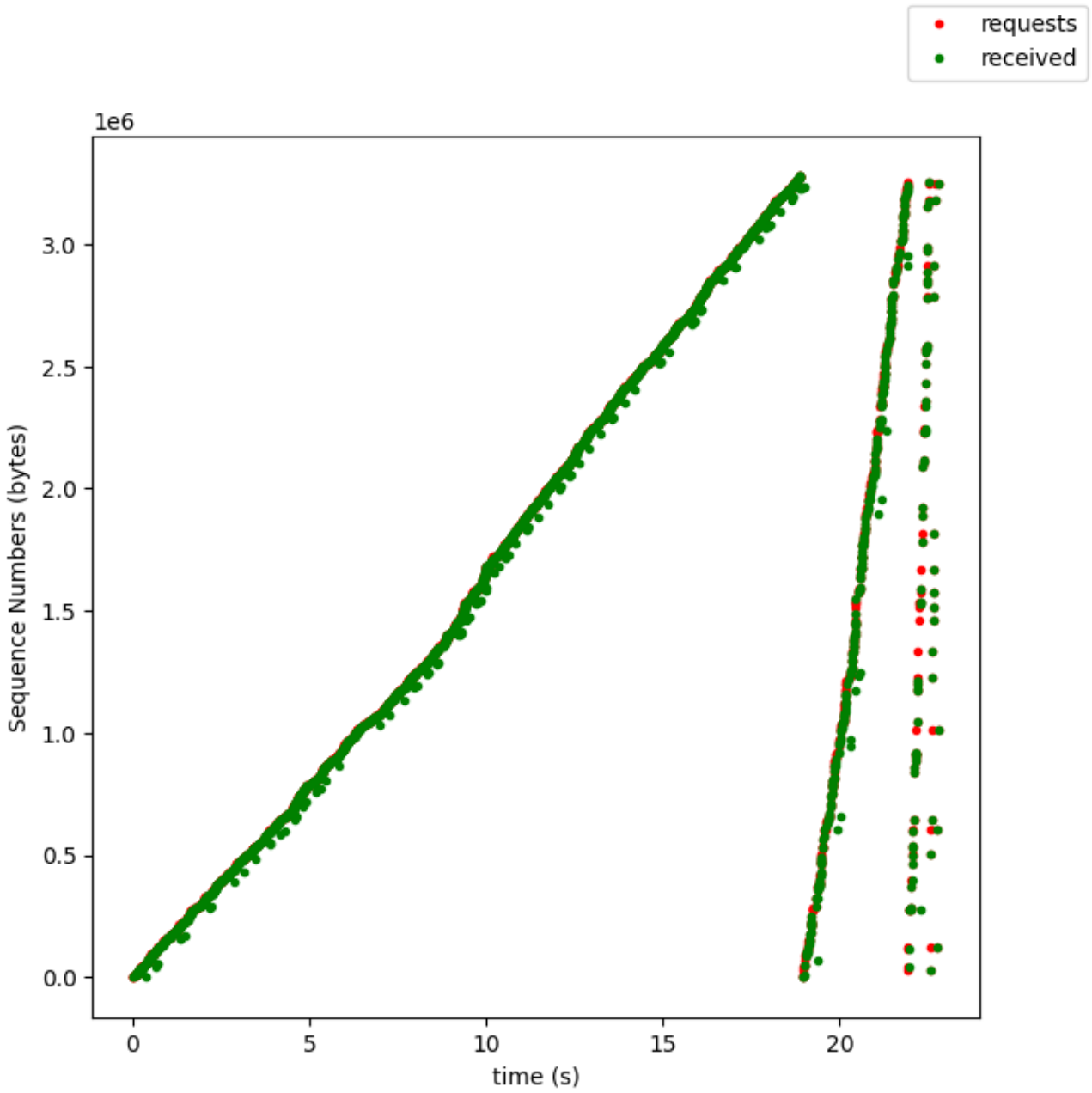


Figure 20: Sequence Numbers in our dynamic implementation for variable rate servers

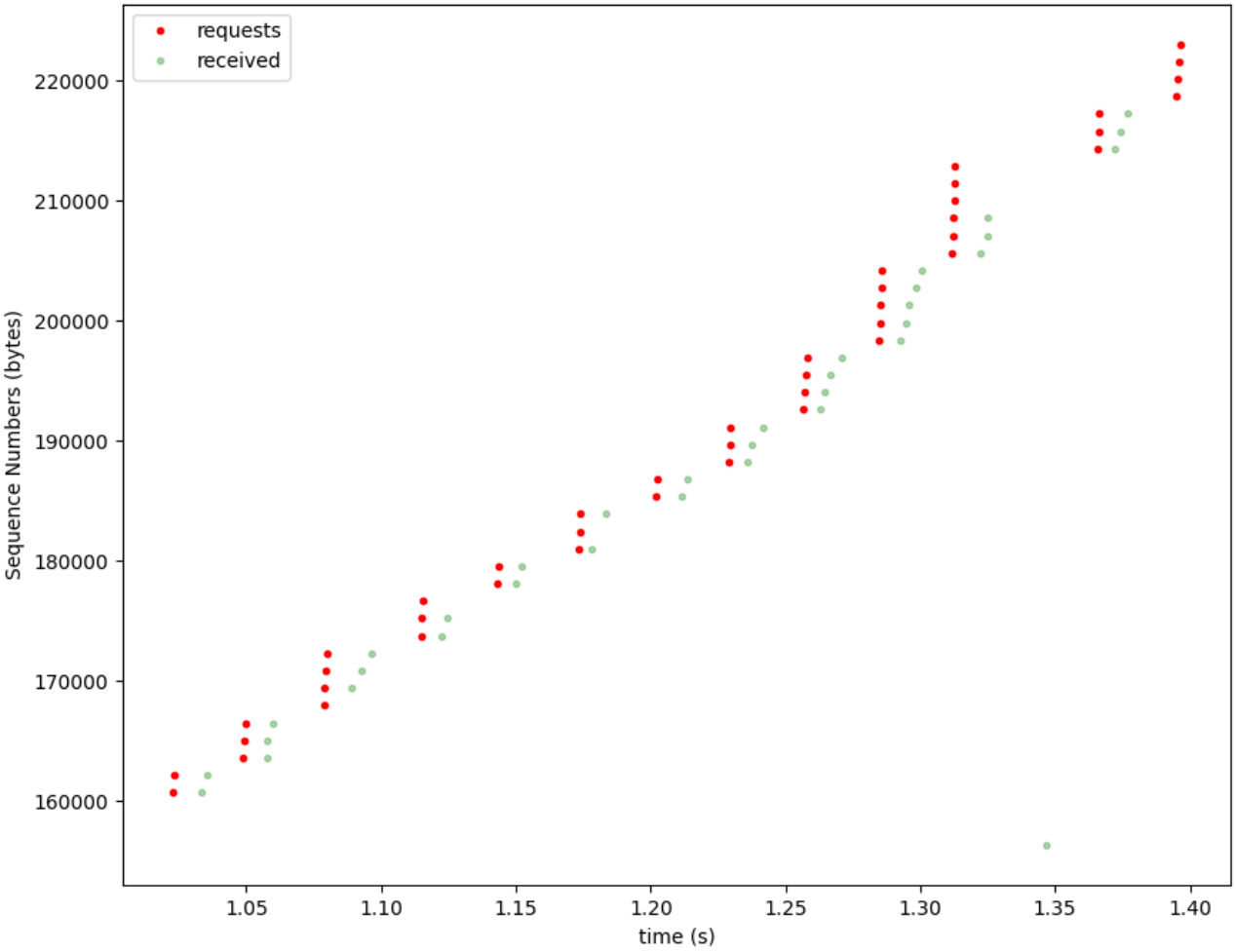


Figure 21: Zoomed in graph of sequence numbers in our dynamic implementation for variable rate servers

9 Observations for Checkpoint 3

Frequency of packets dropped	441	212	77	26	11	1	0	0	0
Number of packets dropped	0	1	2	3	4	5	6	7	8

Penalty for dynamic implementation	45	23	21	11	0
Time Taken	18.34	19.79	25.56	35.56	41.23

- We observed the data of the frequency at which a particular number of packets are being dropped and concluded that most of the cases in which single packet drops are happening are probably due to the random dropping of packets so we can decrease the rate reduction that we are doing for it as compared to the reduction when 2 packets are being dropped, the latter case being more stringent.
- The reason we said that the implementation having constant x has a draw-back is because changes in EWMA are not big enough to slow-down the rate of sending packets and thus it causes squishification when the network is highly congested. This is why we decided to keep a dynamic x, and it precisely prevents this problem.
- Observing the burst-size, squish period graph for constant x we observe that after getting squished, we keep the burst size constant for a large amount of time, which helps it to get unsquished.
- From the graph of burst-size, squish period for dynamic x we can observe that the variance in the burst-sizes is very high because of the fact that we are changing it whenever we need to slow down. Which is in contrast to some of our earlier implementations for constant rate servers.
- We varied various parameters of our dynamic implementation to get the data of time-taken vs penalty obtained without getting squished, we observe something that's expected, a trade-off between penalty and the time taken to submit, both being inversely related to each other.
- From the graph of sequence numbers vs time for the constant x implementation, we can observe the flattening that is happening due to squishing of the network. We can also observe the speedup in the second pass where our implementation is able to get out of the squishing by holding itself back, thus leading to a rapid downloading rate increase.
- We can observe that the sequence number graphs for the dynamic implementation is kind of similar to the constant burst-size case when we are not getting squished as we are essentialy getting most of the data in one pass, and the random dropping of packets remains the same, because of which the curves look similar.

I want to thank the professors and TAs for creating this really interesting and amazing assignment which made us think a lot abous various strategies and introduced us to network programming.