COL334 : Computer Networks

Assignment - 03

Controlling Request Rate and Avoiding Squishing if Variable Token Rate

Aman Singh Dalawat & Mayank Badgotya
2021CS50610 & 2021CS10583

November 1, 2023

Contents

§1 Overview

- In this checkpoint, our goal is to maintain a **dynamic** request rate of sending packets to the vayu server which has a **variable** token regeneration rate.
- The algorithm must also ensure that data is received, stored, and submitted reliably. This reliability is derived from the 1st checkpoint, where the sole goal was to ensure the accuracy of the data.
- We need to maintain a request rate that is dynamically adapting and neither too high such that it causes squishing of the connection, nor it is too low such that it takes very long time to completely receive all data packets.
- Thus we used the approach of **AIMD (Additive Increase Multiplicative Decrease)**. This algorithm maintains a variable **burst size** which is **increased (by 1)** in case we get the expected number of packets and **decreased by a factor of 2** when we receive less than the expected number.
- We also calculated the **RTT** dynamically using **Exponential Weighted Moving Average** technique of TCP.
- Thus our algorithm aims to avoid getting squished by the server by controlling the number of requests it sends as a burst, as well as maintaining the reliable receiving of data while keeping track of RTT.
- Later we compare this approach to **AIAD(Additive Increase and Additive Decrease)** and also the case where burst size is kept constant.
- In the following report, consider **orange** points as request sent and **blue** as receiving points.

§2 AIMD Algorithm

The algorithm uses various techniques, specific to the case where the token regeneration rate of the server is not constant, to achieve the fore mentioned goals. :

§2.1 Initializing parameters and data structures

- Packet size is fixed to **1448 Bytes** (if less than 1448 bytes are remaining then the remaining bytes is requested) which is the maximum allowed size.
- Wait time, which is the server timeout while receiving file size is set to 0.1 seconds.
- Number of packets in the burst window is initialized by 1.
- A dictionary **ack_queue** is initialized which **stores all the packets which are to be requested**.
- A dictionary **file_lines** is initialized which **stores all the packets which were received successfully**.

§2.2 Knowing the size of the file and estimating initial RTT

- Send size request is sent to the server 100 times, which obviously, gives the file size, but also gives an estimate of the time between requesting and receiving of data.
- Average initial round trip time of a packet is estimated by calculating the **median of RTT** of all the send size requests sent. We decided to use the median since it is less affected by outliers.
- Further we used **EWMA** for calculating RTT dynamically.

§2.3 EWMA

- A dictionary **sent_time** maintains the record of time when a request for an offset is sent.
- When we receive a response we update the RTT by taking the difference in current time and maintained time.
- Equation: $RTT_{new} = 0.8 * RTT_{previous} + 0.2 * (time\_sent - time\_received)_{offset}$

§2.4 Requesting packets

- This function repeatedly sends bursts of requests of size N, which is a global variable and keeps updating upon successfully receiving the packets or if the packets get dropped.
- After sending the burst, it calls the receive function to start receiving the packets sitting in the buffer.
- After receiving the packets (whether all or some fraction of N) the receive function returns the number of new packets received.
- Based on the number of requests successfully received, the request function updates N. It is incremented by 1 if the number of requests received is equal to N and decremented by a factor of 2 when the number is not equal to N.
- The function returns when ack_queue becomes empty since when ack_queue becomes empty, it means there are no requests pending to be received. Hence the request function exits.

§2.5 Receiving Packets

- It optimistically waits for N+3 packets. We here are optimistic about delays introduced which is why we are waiting for 3 more packets as each packet request waits for at most 1 RTT. 1 for receiving a response + 2 for delays introduced in the network. It waits for each of the packet for server time out, which is set to be RTT.
- If a packet is not received during the server timeout, then it assumes the packet was dropped. We increase the RTT by 0.5% as we might have exhausted the limit.
- Those requests that were received are removed from ack_queue (hence this dictionary only contains requests which are yet to be received) and added to file_lines (hence this dictionary only contains requests which are successfully received).

- In case the message received indicates that **the server was squished, N is halved**. Also server timeout (which is RTT) is increased by a factor of 1% which over a hundred squish messages roughly doubles the RTT.
- Finally, the function returns the **number of request which were received successfully** (within server timeout).

§2.6 Flushing the buffer

- For some offset if we requested it multiple times. The response will be dumped in the buffer.
- So we **flush** out the **buffer before submitting** our finally downloaded file.

§2.7 Submitting the data

- When we have received all the files, we accumulate all the data from the file_lines dictionary.
- Then we calculate the **MD5 hash** for the accumulated data.
- We send the hash to the server.
- We are waiting for the response from the server. If the response is received we print the response and **in case the response gets dropped we repeat the previous step**.

§3 Why we chose this approach over others?

§3.1 AIMD vs AIAD

- **In the AIAD approach**, when the number of received packets is below a threshold of expected packets it should have received, **we decrease burst size by only 1**.
- In this case, **AIAD will be much more susceptible to squishing than AIMD** because the burst size would not be able to decrease fast enough when token regeneration rate falls quickly.

§3.1.1 Comparing total time in both approaches

Sr. No.	Time Taken	Penalty	Squishes
1	21.919 sec	212	2
2	18.052 sec	200	1
3	22.078 sec	206	2

Table 1 Time taken by AIAD in 3 runs

- As we can see through this data, AIAD has **higher penalty** than AIMD.
- AIAD is running **'close to the edge'**, thus it is getting more squished as compared to AIMD, when token generation rate is very less.

Sr. No.	Time Taken	Penalty	Squishes
1	20.374 sec	79	0
2	20.451 sec	86	0
3	19.863 sec	73	0

Table 2 Time taken by AIMD in 3 runs

§3.1.2 Burst Size vs Time in both cases

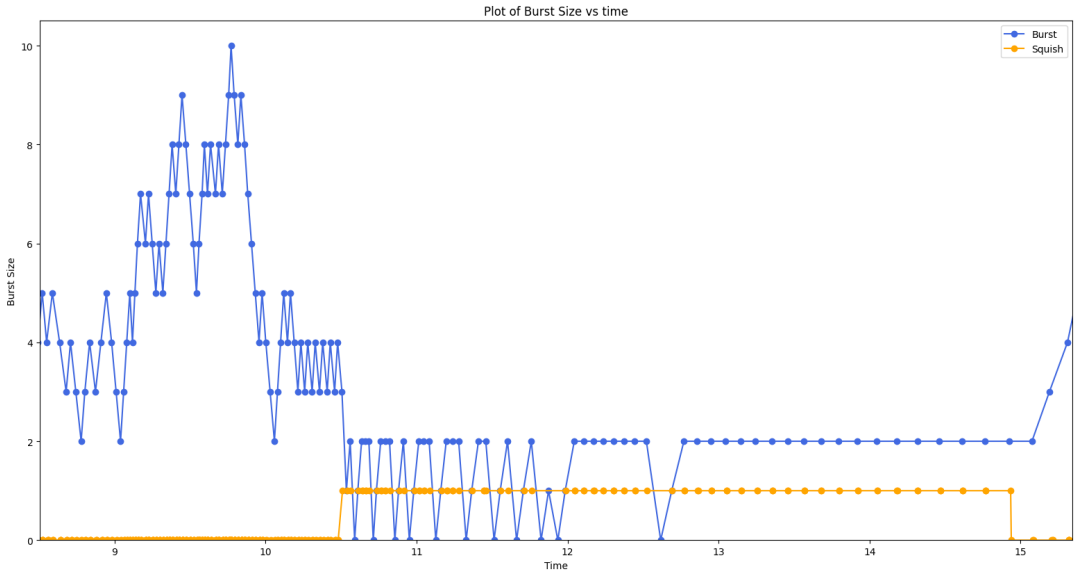

Figure 1: Burst Size vs Time for Vayu server using AIAD

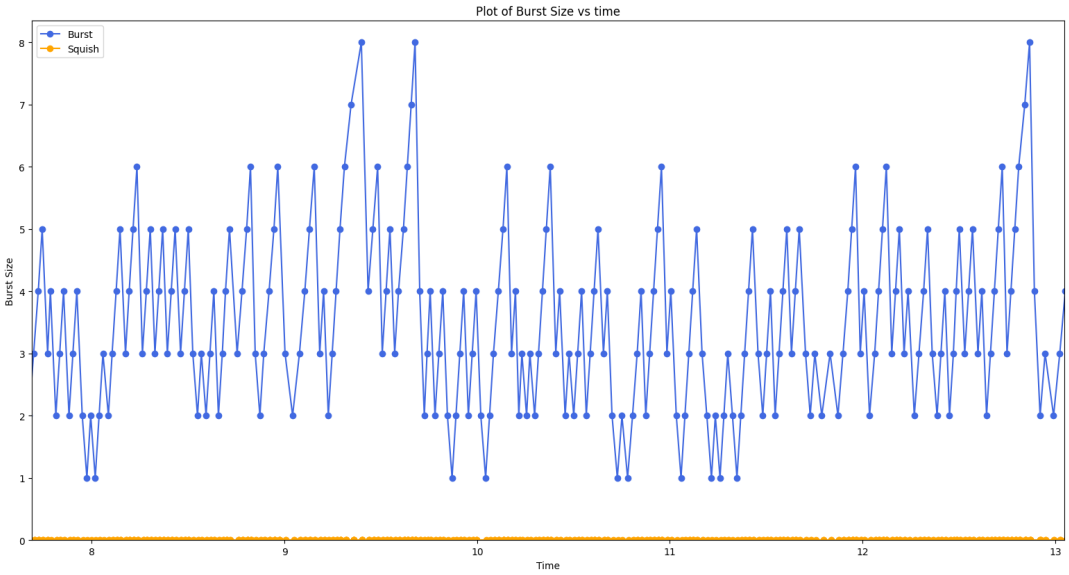

Figure 2: Burst Size vs Time for Vayu server using AIMD

- In the case of AIAD, we can see when number of packets received were less than expected the burst size decrease by only 1. The lowest burst size it reached was 0.

- In the case of AIMD, we can see when number of packets received were less than expected the burst size becomes half. The lowest burst size it reached was 1.
- We can see that AIMD doesn't get squished and try to utilise the full rate while AIAD gets squished.

§3.2 Fixed vs Dynamic Burst Size

- In the fixed burst size approach, we fixed a certain burst size (N = 7 and N = 4).
- We saw N = 4 and N = 7 performed randomly. As fixing N is not scalable we refrained from using fixed N.

§3.2.1 Comparing total time for fixed burst sizes

Sr. No.	Time Taken	Penalty	Squishes
1	20.701 sec	161	1
2	18.585 sec	98	0
3	19.096 sec	126	1

Table 3 Time taken by Fixed Burst Size = 4 in 3 runs

Sr. No.	Time Taken	Penalty	Squishes
1	17.408 sec	299	2
2	17.750 sec	338	3
3	17.263 sec	359	3

Table 4 Time taken by Fixed Burst Size = 7 in 3 runs

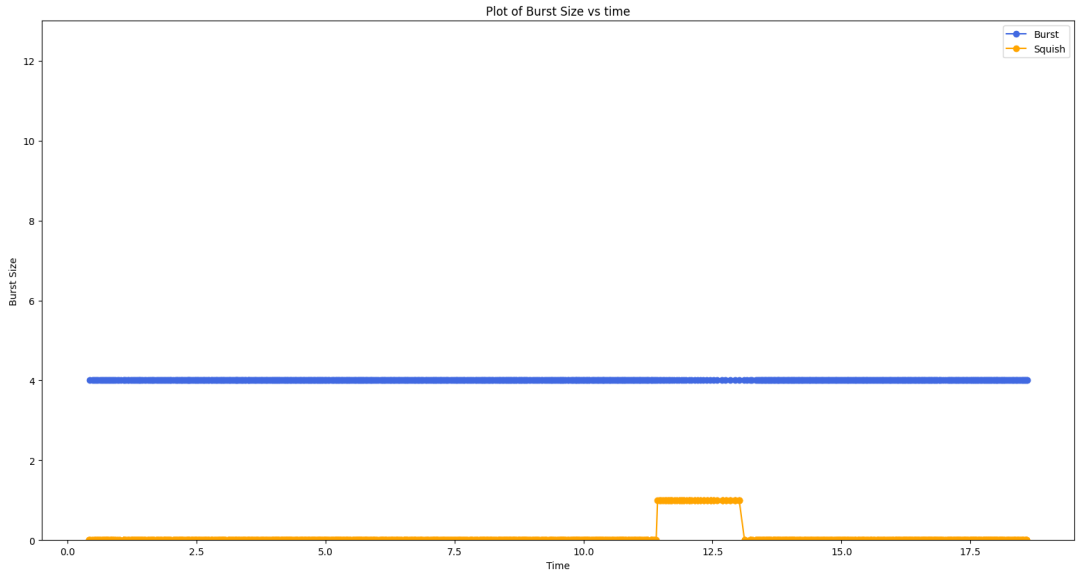

Figure 3: Burst Size vs Time for Vayu server with N = 5

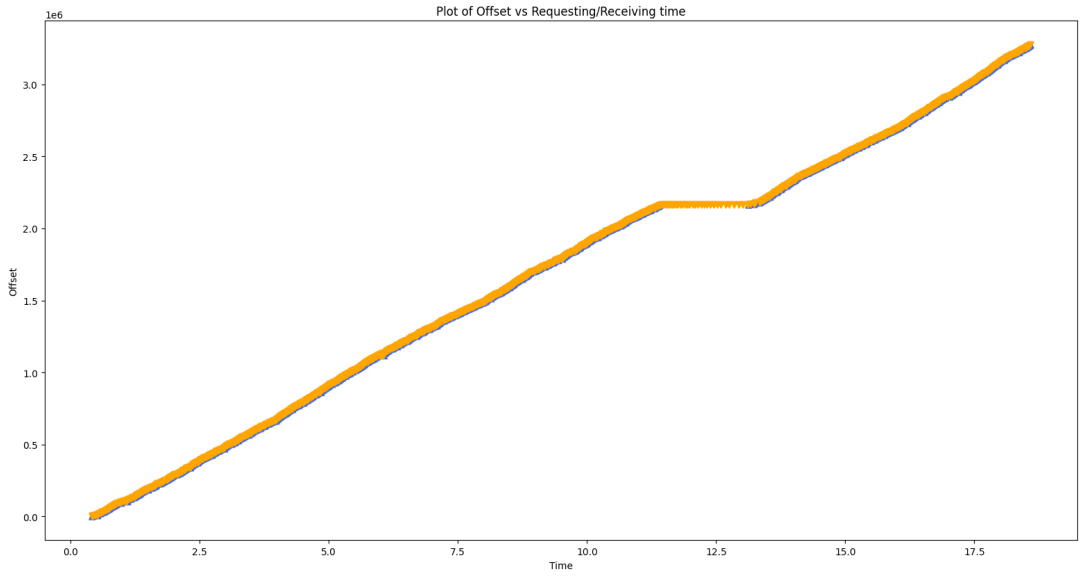

Figure 4: Offset vs Time for Vayu server with N = 5

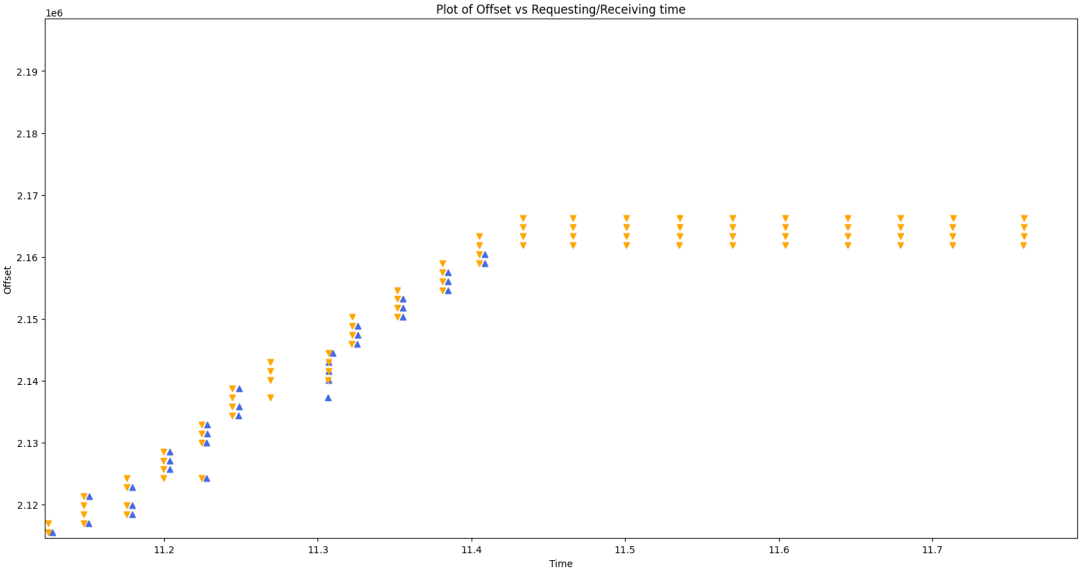

Figure 5: Zoomed offset vs Time for Vayu server with N = 5 at squish

- We can see that fixed burst size exhausts the token when the token generation rate drops. Which introduces cascading squishes.
- If token generation rate had a lower bound then the approach might find a optimal N where squished will be minimized.

§4 Plots for Visualizing the Algorithm

In this section we show plots of offset vs time and burst size vs time and draw observations from them.

§4.1 Offset vs Time on Vayu server and localhost

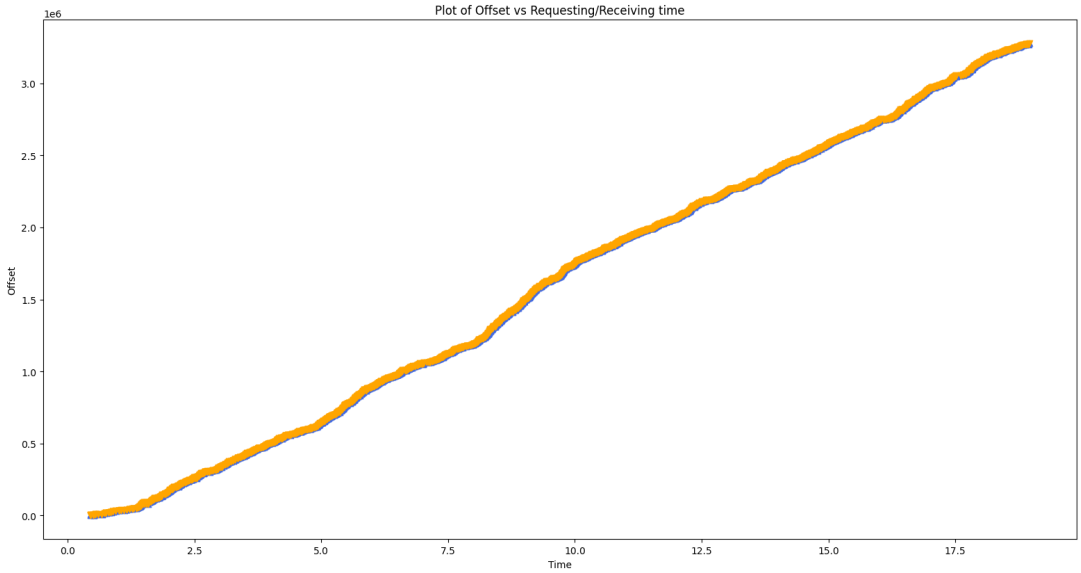

Figure 6: Offset vs Time for Vayu server

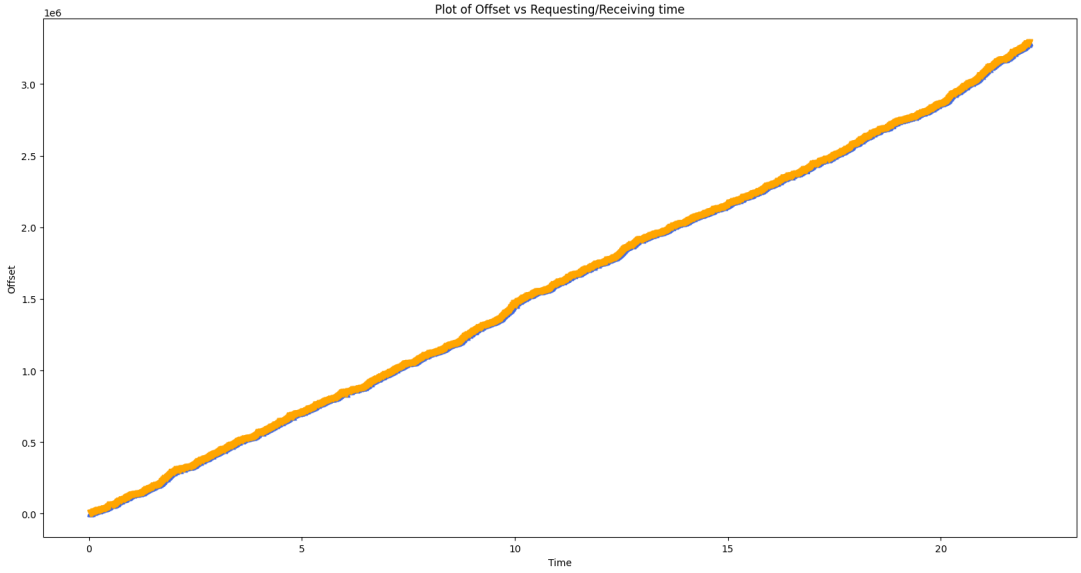

Figure 7: Offset vs Time for localhost

- Since we immediately request for requests that were not received in the next burst (not the round-robin fashion), we observe the graph above, where the request time and receive time almost overlap.
- We saw a much cleaner plot for localhost as it doesn't suffer high traffic and RTT is also low for it.

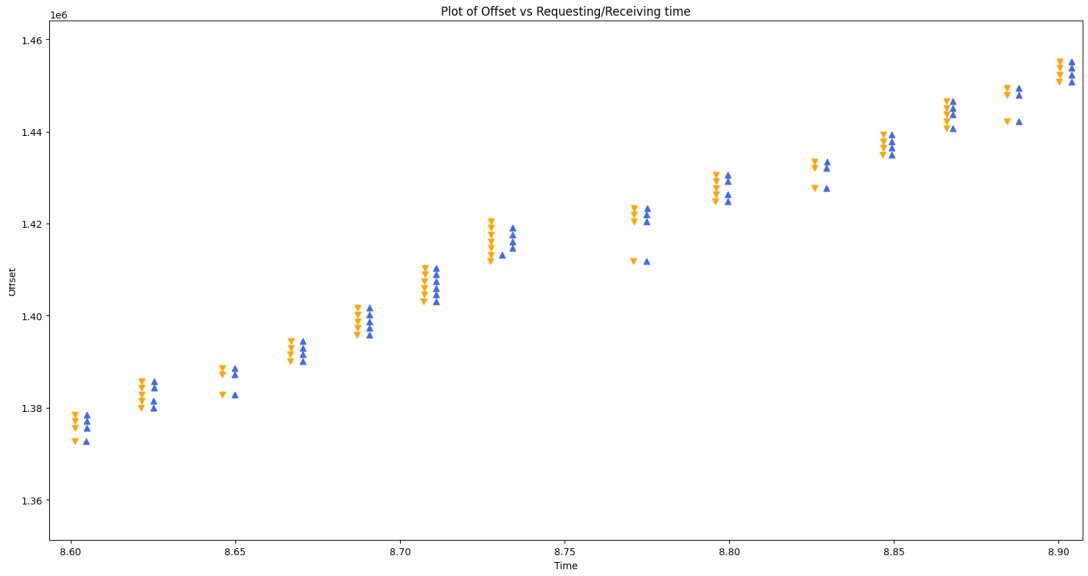

Figure 8: Zoomed Offset vs Time for Vayu server

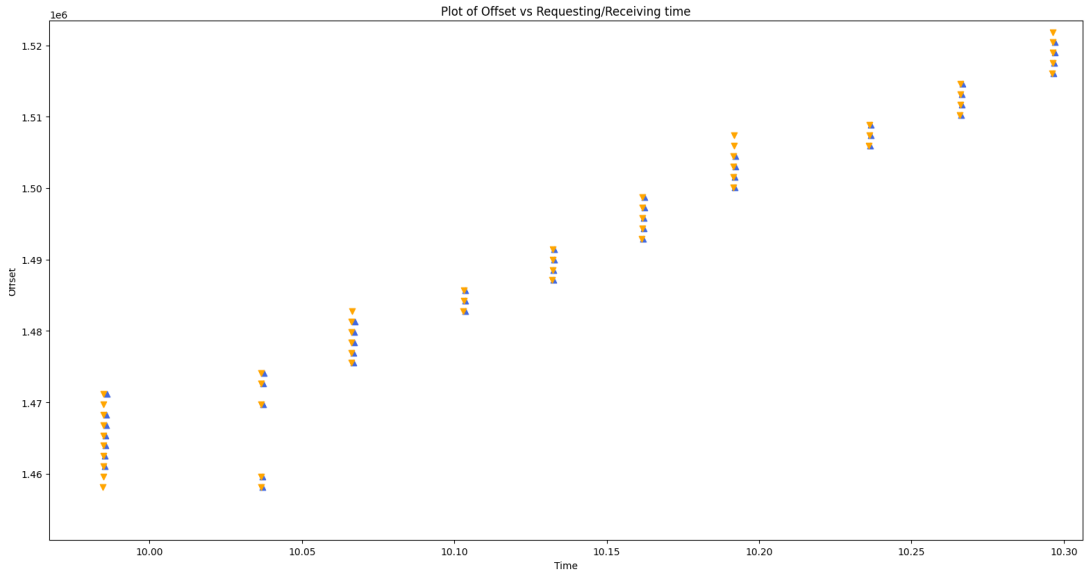

Figure 9: Zoomed Offset vs Time for localhost

- We can see AIMD working in these graphs as burst size changes.
- From the graph we can verify, the distance between the request sent and the response is proportional to RTT.
- For the Vayu server, the distance is variable and significant while in the localhost it is insignificant.

§4.2 Burst Size vs Time on Vayu server and localhost

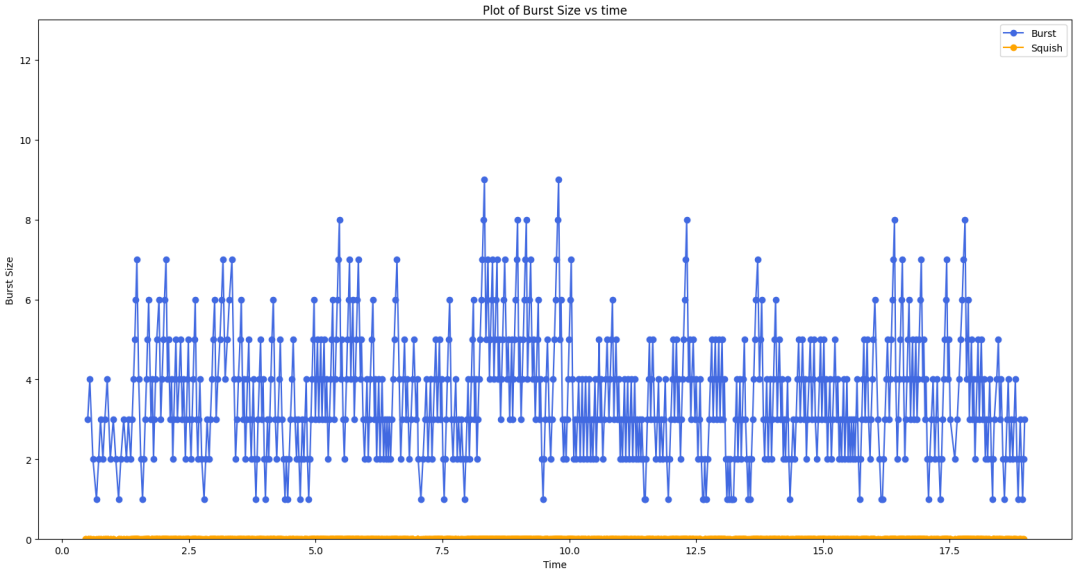

Figure 10: Burst Size vs Time for Vayu server

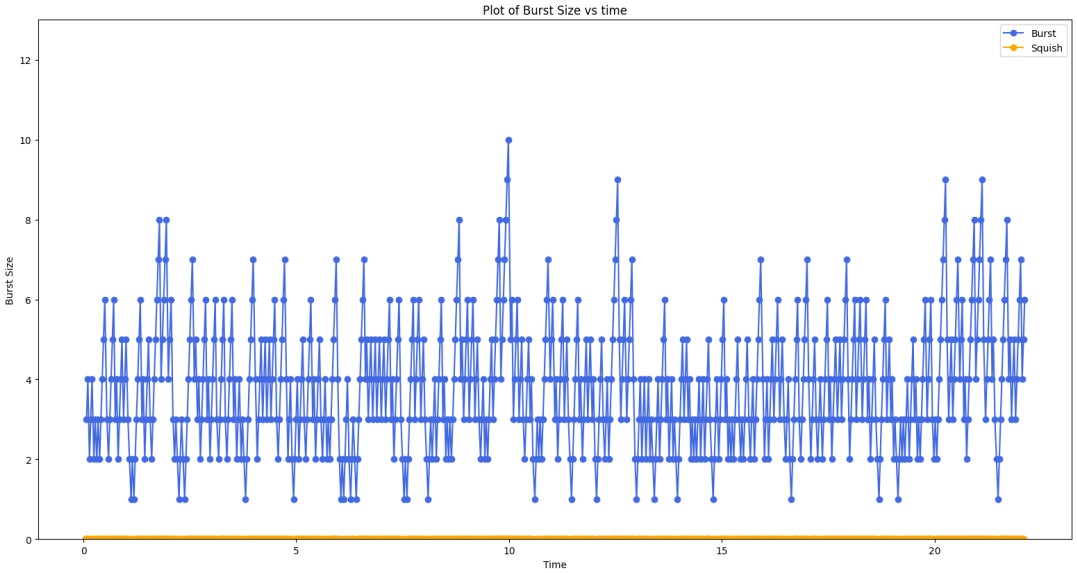

Figure 11: Burst Size vs Time for localhost

- We can see in these plots the typical **saw tooth fashion** of AIMD approach.

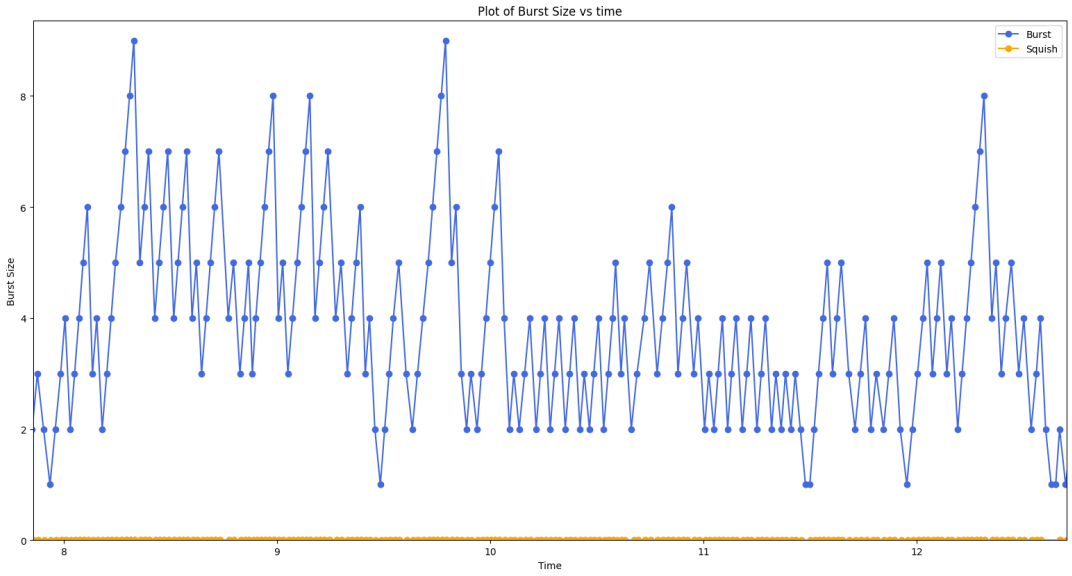

Figure 12: Zoomed Burst Size vs Time for Vayu server

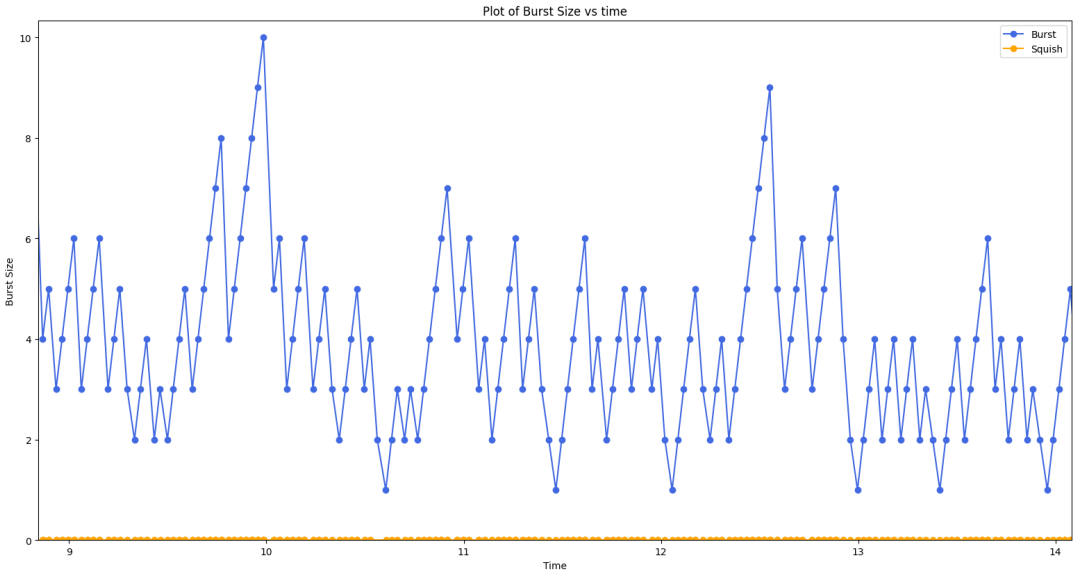

Figure 13: Zoomed Burst Size vs Time for localhost

- These are the zoomed in plots of the above graphs. We can see the slow increment of burst size and abrupt decrease of burst size in case some requests were not received.

§4.3 Estimated RTT on Vayu server

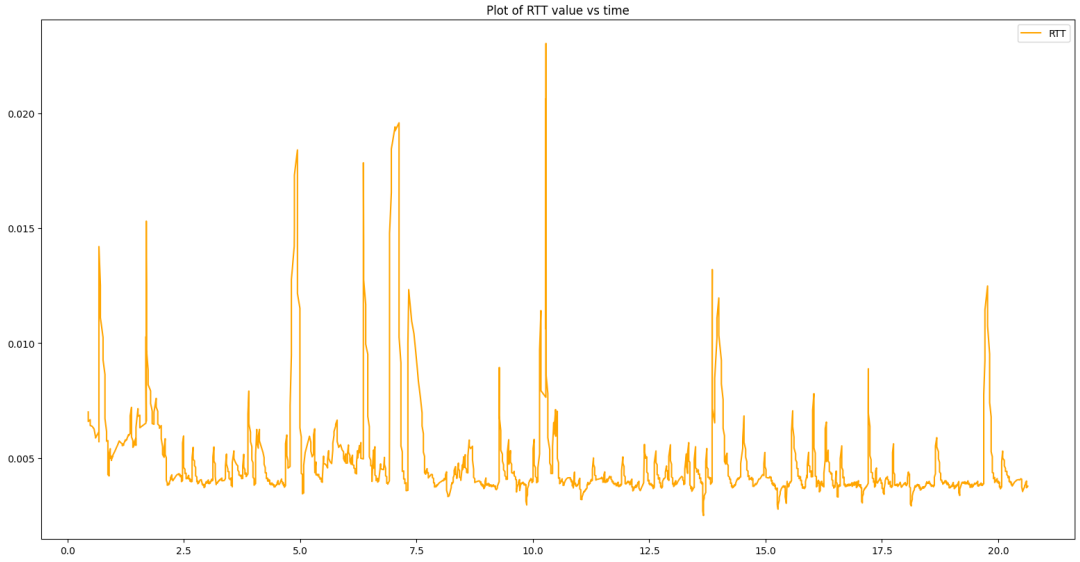

Figure 14: Estimated RTT vs Time

- Since the RTT is calculated using EWMA approach, we can see some peaks and valleys in the graph due to dynamic nature of network.
- We saw that the average RTT is close to 4.5 ms.

§4.4 Conclusion

In conclusion, we can see AIMD with EWMA is a very effictive approach to deal with the scenario of downloading a file from a variable rate server. It is able to work correctly even in the case of large server traffic by maintaing a variable estimate of RTT. This algorithm meets the following expectations:

- The algorithms maintains the correctness of data, as it is of the utmost importance that data does not get corrupted while receiving from server.
- The algorithm maintains a dynamic burst size which is sensitive to packet drops from the server. This ensures that squishing is avoided as squishing can heavily delay the receiving of file.
- The algorithm also favour the increment of burst size when it observes that all the requested packets are received. Thus it pushes to keep the burst size as large as possible.