COL334 : Computer Networks

Assignment - 03

Controlling Request Rate and Avoiding Squishing if Constant Token Rate

Aman Singh Dalawat & Mayank Badgotya
2021CS50610 & 2021CS10583

October 25, 2023

Contents

§1 Overview

- In this checkpoint, our goal is to maintain a dynamic request rate of sending packets to vayu server which has a constant token regeneration rate.
- We need to maintain the request rate which is 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.
- The algorithm must also ensure that data it received, stored and submitted reliably. This reliability is derived from the 1st checkpoint, where the sole goal was to ensure accuracy of the data.

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 expected number.

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 it maintain the reliable receiving of data.

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 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 round trip time of a packet is estimated by calculating the **median of RTT** of all the send size requests sent.
- Initially we used the mean of the RTT's, but since it was fluctuating a lot due to some exceptionally high or low values. Thus we decided to use median since it is less affected by outliers.
- Finally server timeout is set to be the maximum of the calculated RTT and 0.005 (this is done as a precautionary measure to avoid squishing in case RTT is too small and token rate is also small).
- This calculated RTT is used further to calculate the time gap between two consecutive burst windows.

§2.3 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 **waits for 3RTT** and then calls 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 packets received.
- Based on the number of requests successfully received, the request function updates N. It is incremented by 1 if number of requests received is more than a certain threshold and decremented by a factor of 2 when number is below the threshold.
- 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.

§2.4 Receiving Packets

- This function expects to receive N packets, which is the current burst size. 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 droppped.
- Those requests which 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 1.01 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).

§2.5 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.

§2.6 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 contrast to AIMD, in this approach (AIAD) the rate of decrease will be slow compared to AIMD**.
- Since the rate of token regeneration is constant on 9801 server, when we increment the burst size, it will never be much larger than the token generated per second.
- Thus we can expect **AIAD** to be somewhat **faster than AIMD**.
- But problems will occur when token regeneration rate is very less and RTT is also less.
- In this case, **AIAD will be much more susceptible to squishing than AIMD** because between 2 burst, the wait time will be less and tokens will not be generated fast enough.

§3.1.1 Comparing total time in both approaches

Sr. No.	Time Taken	Penalty	Squished
1	13.935 sec	18	No
2	12.771 sec	65	No
3	13.758 sec	59	No

Table 1 Time taken by AIAD in 3 runs

- As we can see through this data, **AIAD actually runs faster than AIMD, but penalty is very high** in case of AIAD compared to AIMD.

Sr. No.	Time Taken	Penalty	Squished
1	19.373 sec	14	No
2	17.414 sec	9	No
3	17.808 sec	10	No

Table 2 Time taken by AIMD in 3 runs

- AIAD is very 'close to the edge', thus it is very susceptible to getting squished as compared to AIMD, when token rate is very less.

§3.1.2 Burst Size vs Time in both cases

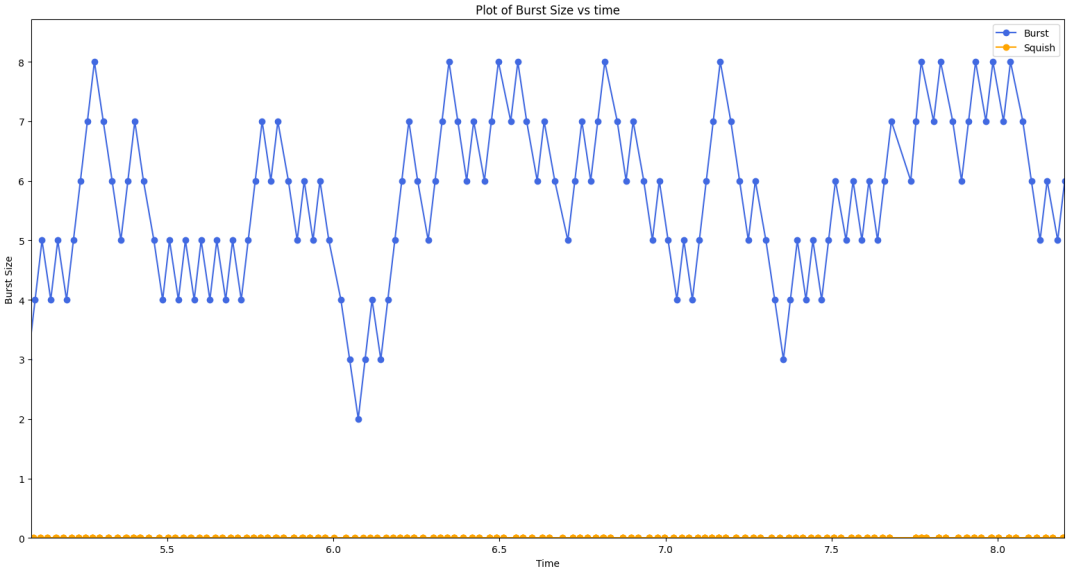

Figure 1: Burst Size vs Time for Vayu server using AIAD

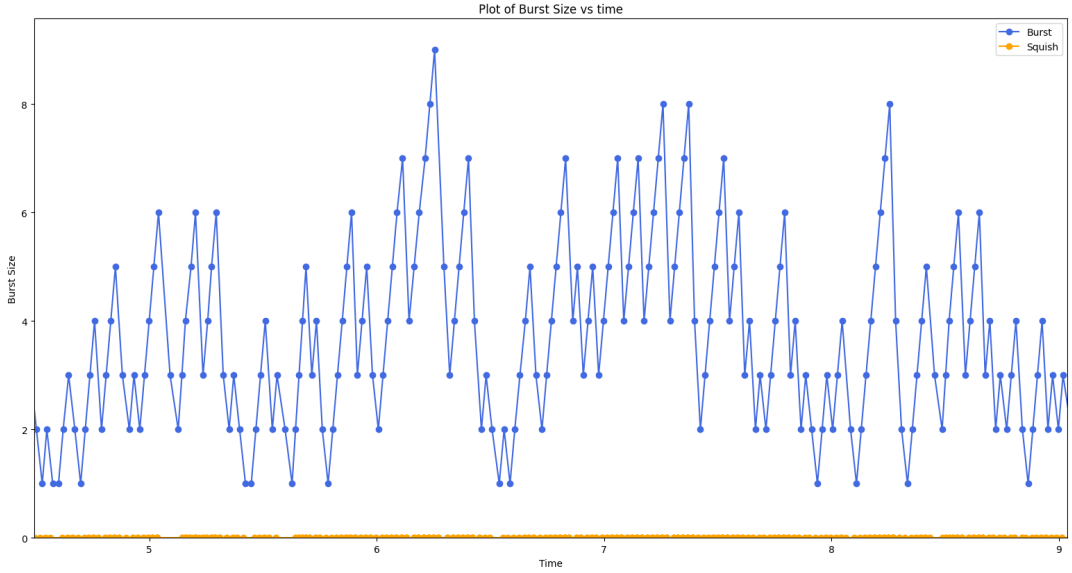

Figure 2: Burst Size vs Time for Vayu server using AIMD

- 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 2.
- 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.

§3.2 Fixed vs Dynamic Burst Size

- In the fixed burst size approach, we fixed a certain burst size (N = 7) by trial and error and observed better time and penalties than our AIMD approach.
- But this approach will fail even if there is slightly different token rate as more than 7 token rates will definitely cause the server to squish the client repeatedly and the total time and penalties will suffer badly.
- Since this approach is **less scalable**, we decided not to use this approach for the sake of generalization.

§3.2.1 Comparing total time for fixed burst sizes

Sr. No.	Time Taken	Penalty	Squished
1	11.445 sec	158	Yes
2	12.586 sec	277	Yes
3	12.081 sec	216	Yes

Table 3 Time taken by Fixed Burst Size = 8 in 3 runs

- We can see from the data that for **burst size 8, the client gets squished** and the time is increased as compared to burst of size 7.

Sr. No.	Time Taken	Penalty	Squished
1	11.445 sec	31	No
2	11.507 sec	33	No
3	11.442 sec	54	No

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

- For burst size 7, we can see time and penalty incurred are very good. Hence it clearly performs better than our approach in this case.
- But this will not generalize to any other token rate. We can see even changing the burst size from 7 to 8 can result in squishing. Hence this method is not scalable.

§3.2.2 Burst Size vs Time for Burst Size 8 and 7

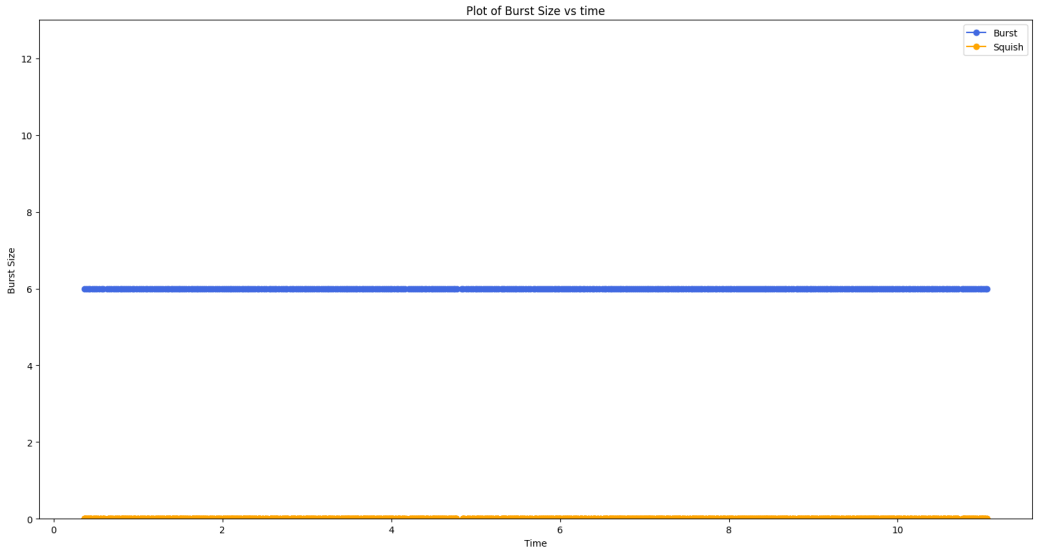

Figure 3: Burst Size vs Time for Vayu server using N = 7

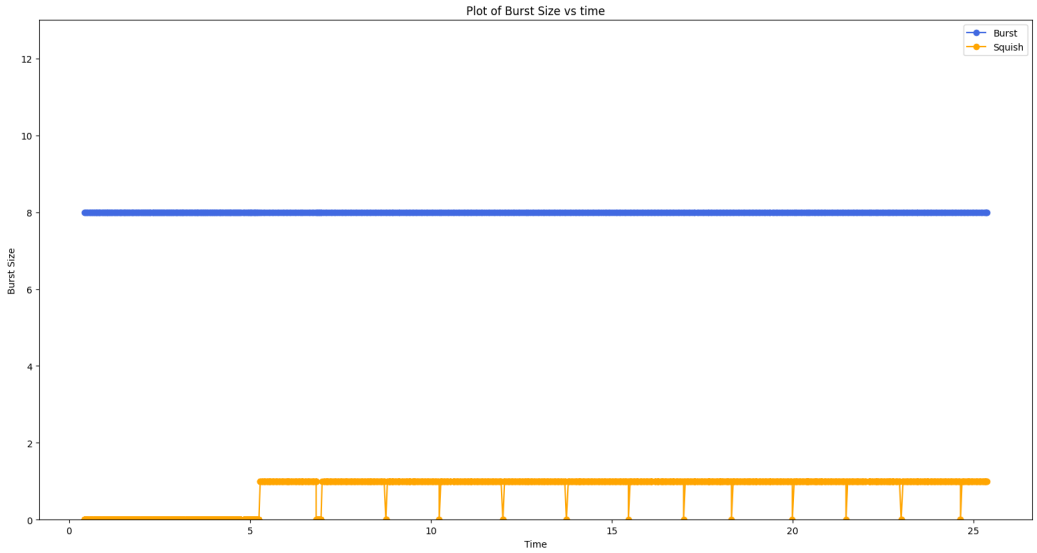

Figure 4: Burst Size vs Time for Vayu server using N = 8

- We can see no squishing in case of N = 7 and the burst size is constant.
- Squishing occurs multiple times in case of N = 8 and burst size is constant.

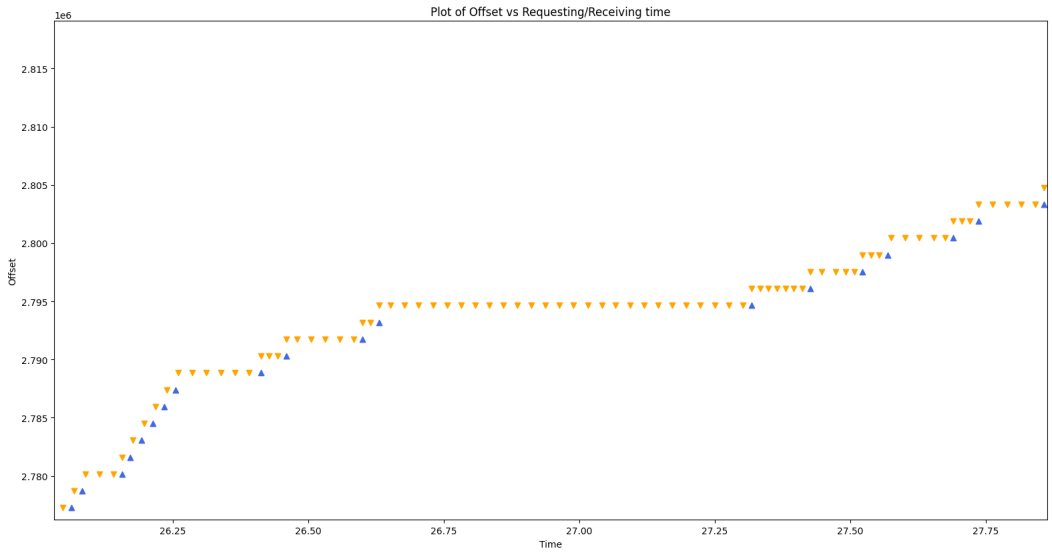

Figure 5: Offset vs Time for Vayu server using N = 8

- Since the client got squished, a large number of requests were dropped by the server, hence continuous streak of yellow tags.

§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

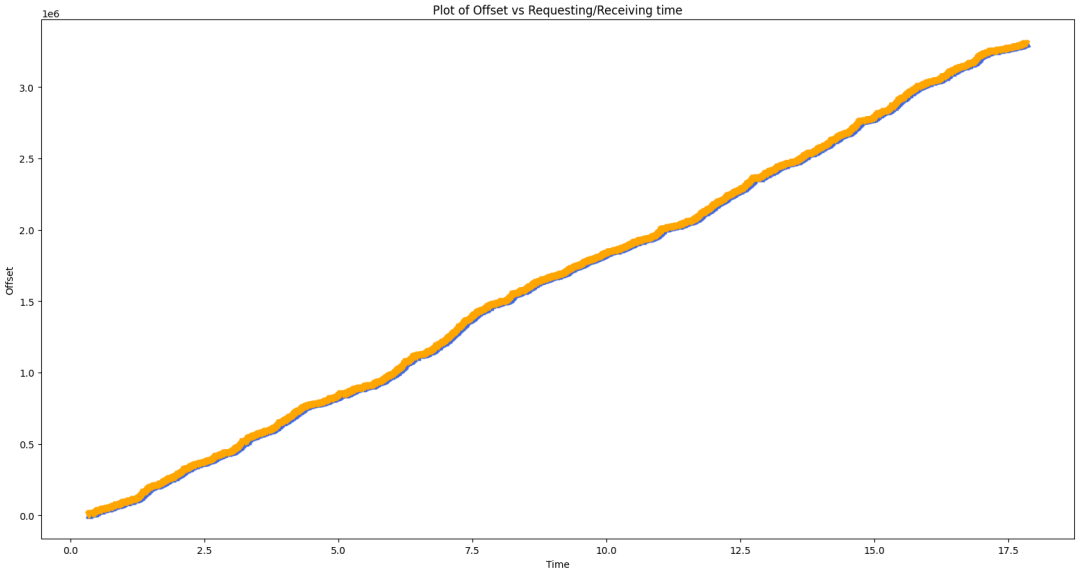

Figure 6: Offset vs Time for Vayu server

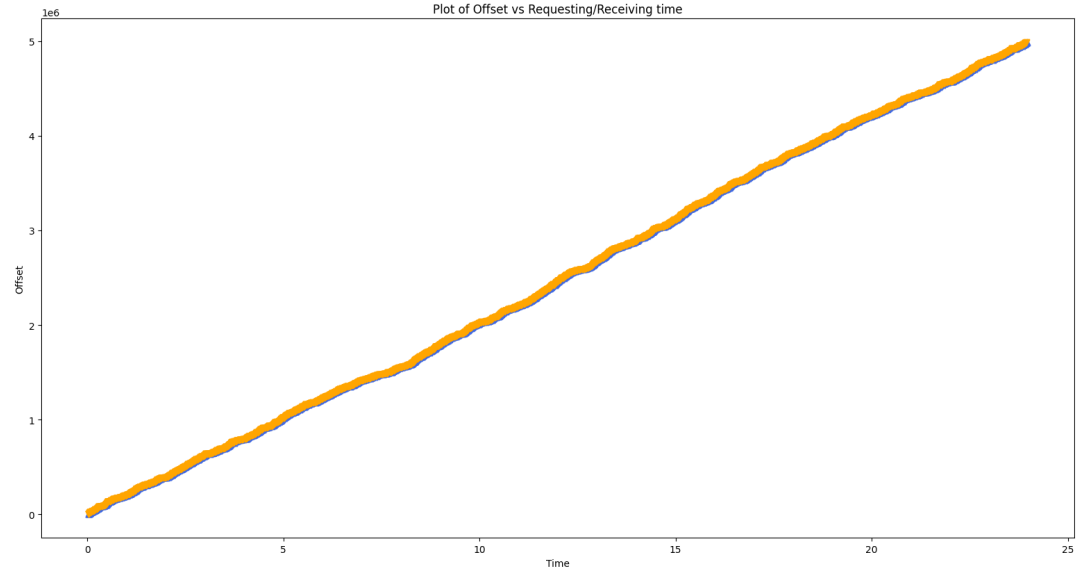

Figure 7: Offset vs Time for localhost

- Since we immediately request for requests which 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 overlaps.

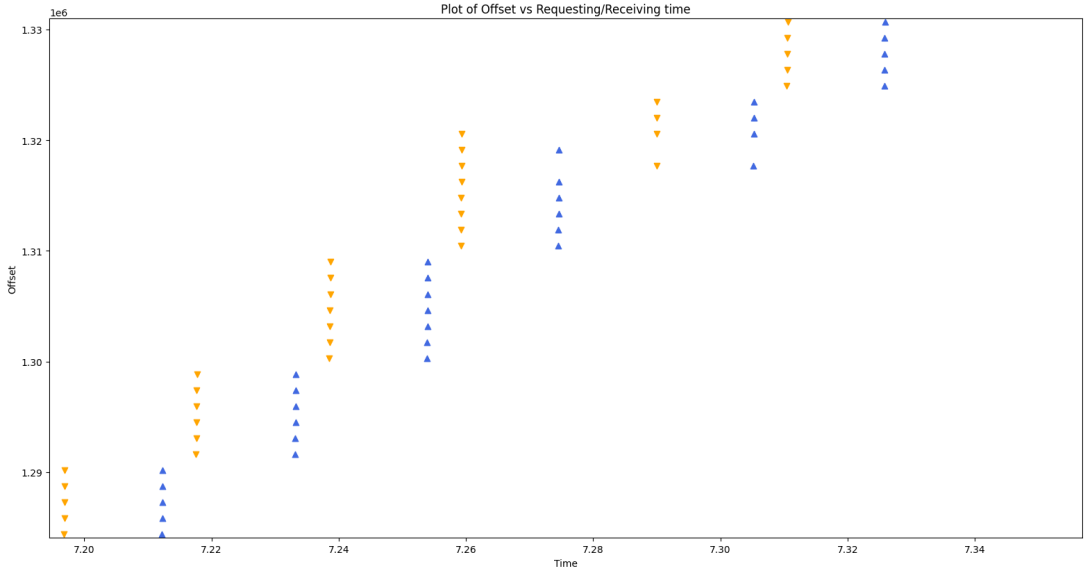

Figure 8: Zoomed Offset vs Time for Vayu server

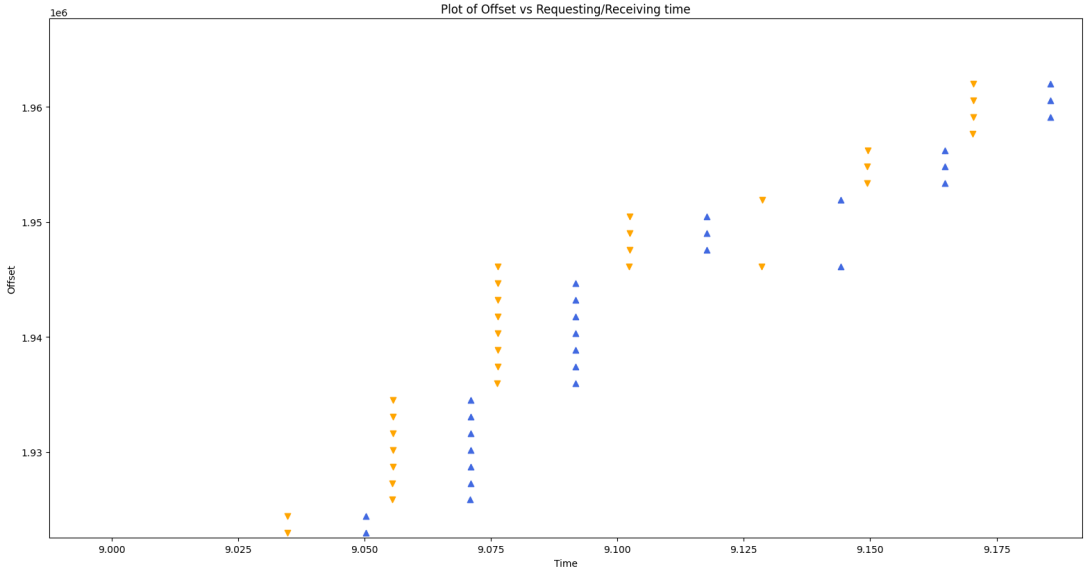

Figure 9: Zoomed Offset vs Time for localhost

- This is a zoomed version of the above offset vs time graph. We can see when only 6 out of 7 requests were received successfully, burst size is halved to 4 and the offset which was not received earlier is requested in the next burst.

§4.2 Burst Size vs Time on Vayu server and localhost

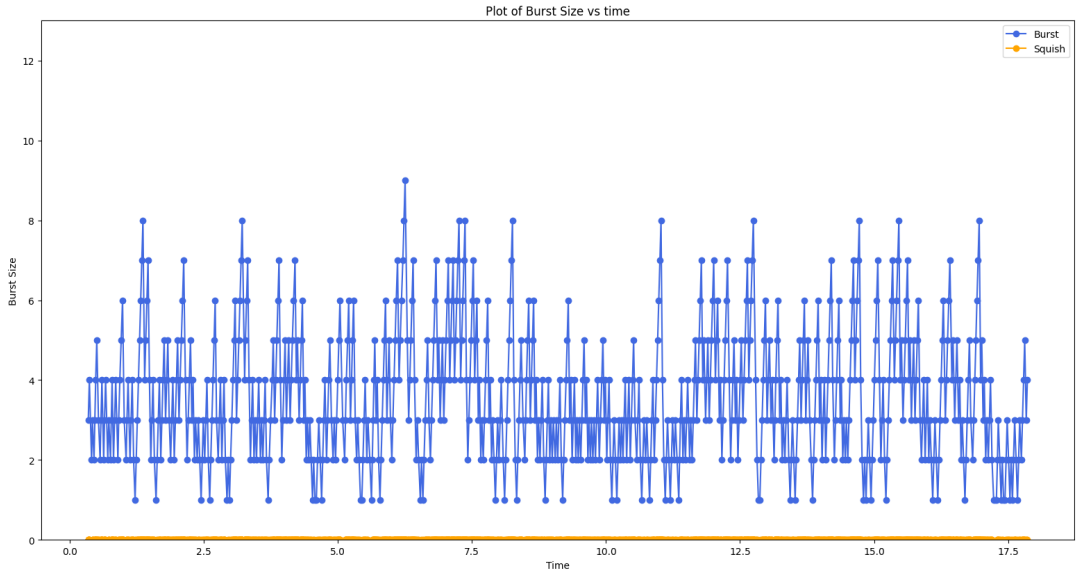

Figure 10: Burst Size vs Time for Vayu server

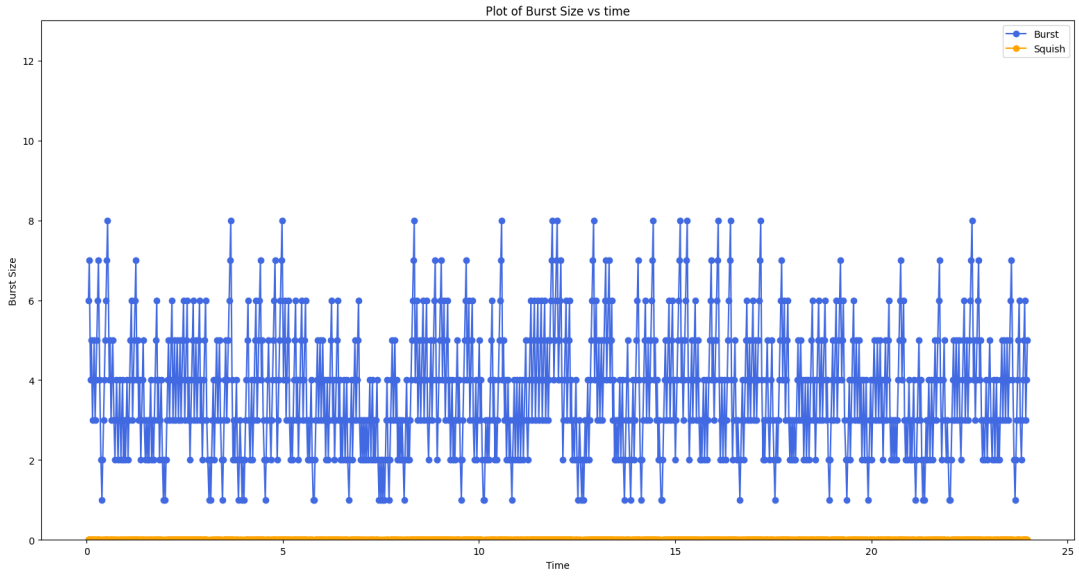

Figure 11: Burst Size vs Time for localhost

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

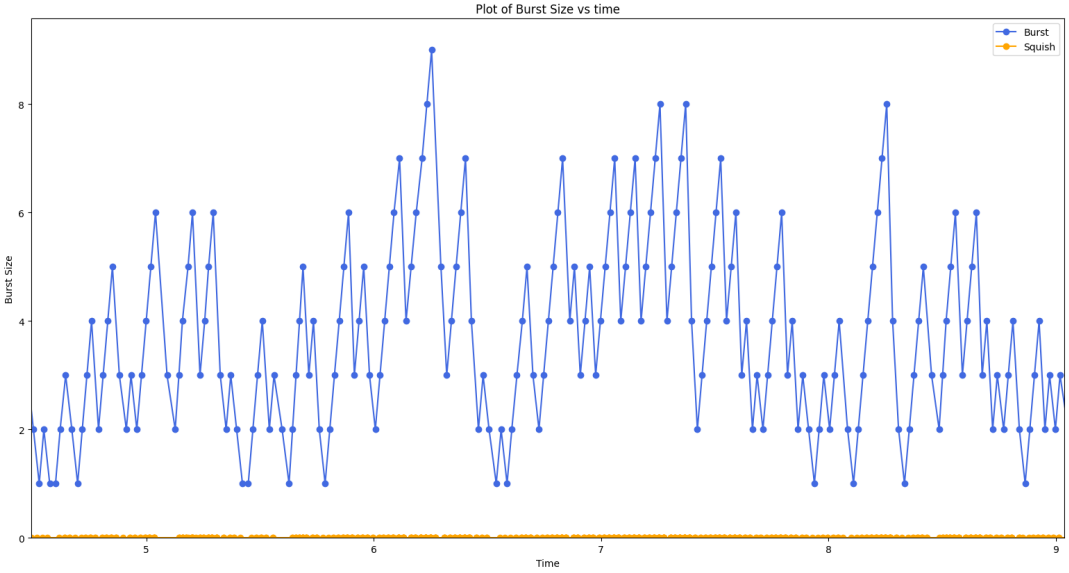

Figure 12: Zoomed Burst Size vs Time for Vayu server

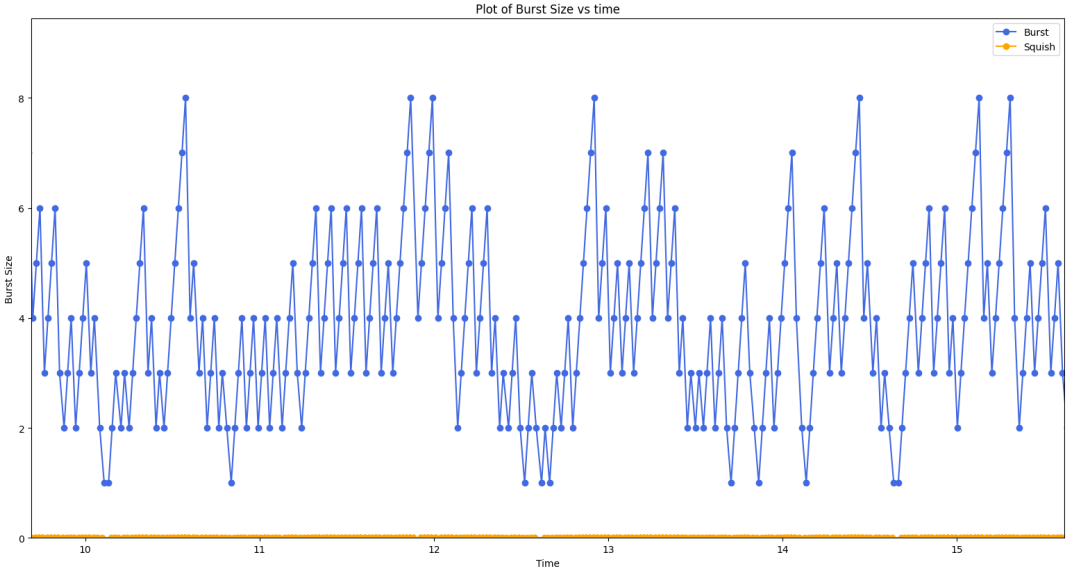

Figure 13: Zoomed Burst Size vs Time for localhost

- 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.