# COL334 Assignment 3 Report
# Part 3

Mrunal Kadhane - 2021CS10109
Kanishka Gajbhiye - 2021CS50131

## 1 AIM

The aim of part3 of the assignment was to figure out how fast to request for data on top of receiving it reliably as to not get squished yet not go too slow on a **variable rate server**

## 2 ALGORITHM

We are operating **AIMD** using the following threads:

1. send-handler : sends data request from a specific offset.
2. rec-handler : receives data from the server simultaneously for the requests sent.

### 2.1 Thread 1 : Send-handler

This thread first sends a burst of 5 into the network. Upon receiving a reply, it pushes another packet into the network. So basically there are **multiple unacknowledged packets** inside the network. Every time a packet is received, the burst size is increased by **1/burst** and once all packets equal to the number of burst are received, the client pushes an **extra packet** into the network and **increases the burst by 1**. Next time this thing repeats, **2 extra packets** are pushed and **so on**. By this way, we are steadily increasing the number of unacknowledged packets between one round trip time. The waiting time between each packet is **(1/send-rate)**.

### 2.2 Thread 2 : Rec-handler

This thread simultaneously receives replies of the send requests. It decodes and adds the line to a dictionary. It performs a check for squished or not squished. If **squished**, it drops down the **send rate along with the burst size to 1**. Otherwise, it **increases the send-rate by 1 and burst size by 1/burst**. Whenever there is a **change of burst size by 1**, it unlocks the loop which pushes and **extra unacknowledged packet (sort of like a burst)** in the send thread. Once we receive a reply, we **increment our offset**. The packets can get reordered on their way back and even dropped. Reordering doesn't affect us. For **packet drops, we just increment our offset**. For the variable rate server, we are continuously checking for squish and hence adjusting our rates and size accordingly. This makes the algorithm more fluid. After the send thread ends, the fill gaps function sends offset requests for missing lines until we have received all the data.

On the next page, we have demonstrated our algorithm through an example. As we see, packet 2 and packet 5 are dropped but they will be recovered by the fill-gaps function. Whenever a reply is received, another offset request is sent. When an aggregate of 5 replies are received, the client pushes a burst of 2 hence increasing the number of unacknowledged packets in the network. The second figure demonstrates the same algorithm with reordering.

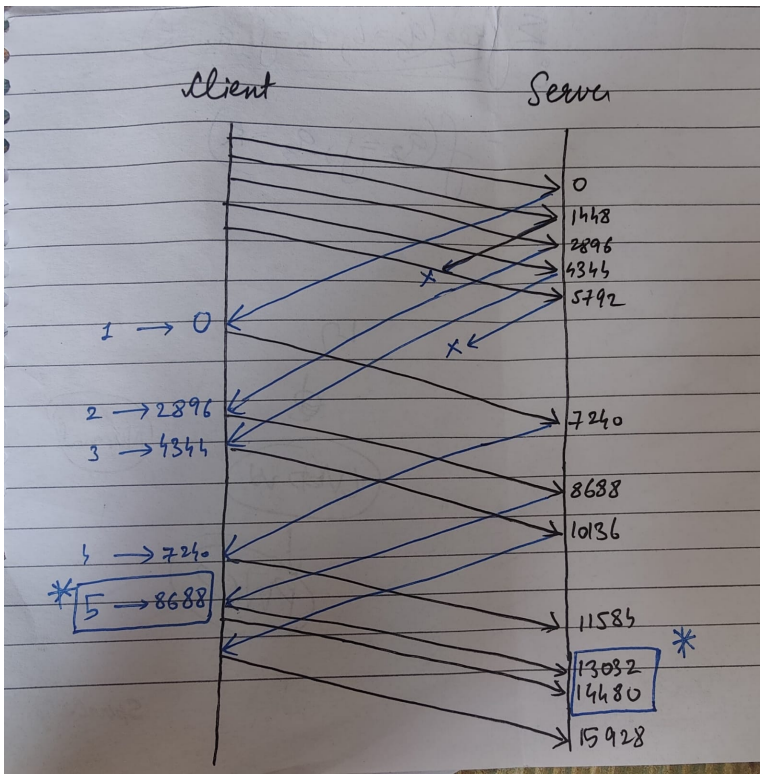


Figure 1: AIMD approach demonstration with initial burst of 5

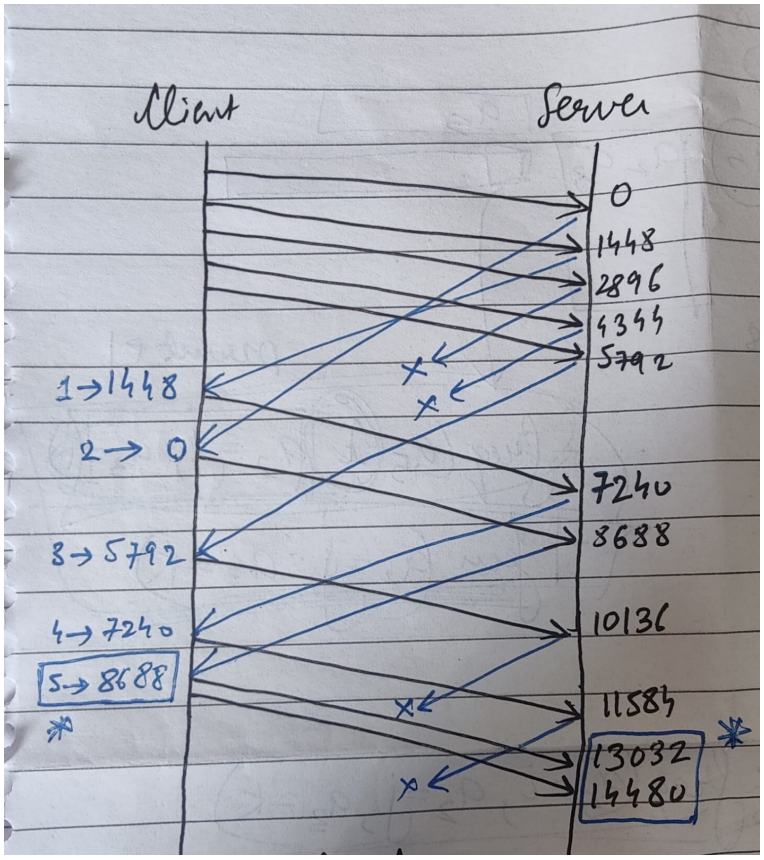


Figure 2: AIMD approach demonstration with initial burst of 5 and reordering

# 3 Graphs and Analysis

Below are the plots for increments, send rate and burst size wrt time.

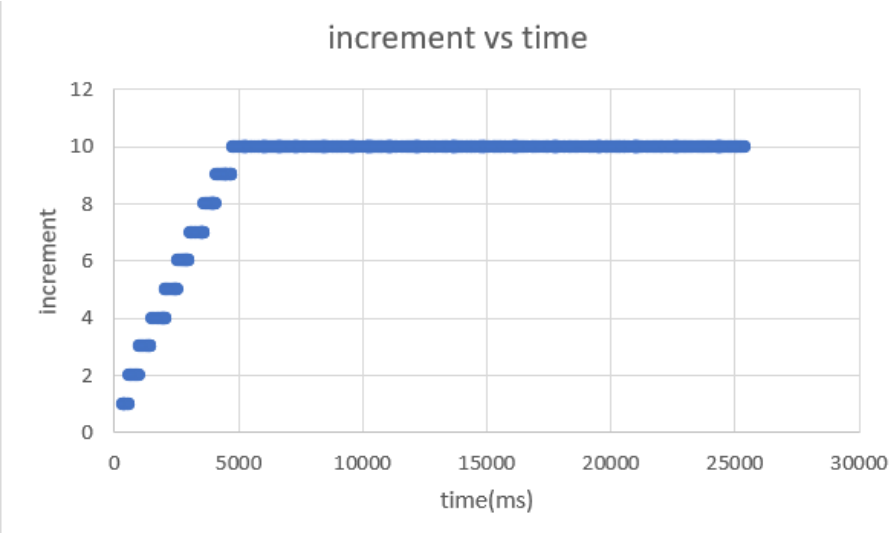


Figure 3: Variation of increment with time

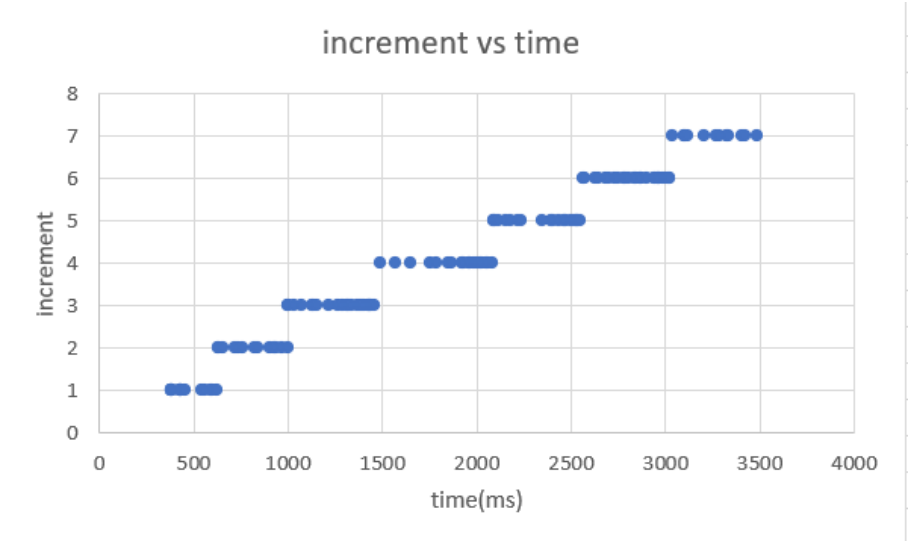


Figure 4: Zoomed-in view of Variation of increment with time for first 4000ms

**Increment** is a variable in our client program which is used to **increase the number of unacknowledged packets** which we are sending to the server after the total number of burst packets are received. It is initialised to be 1. Increment always increases since we never get squished via this implementation. We are setting an upper bound of 10 on increment because without that we have observed that the packets get squished. What can be a possible reason is that when the **increment value reaches a high number, the client sends that number of packets (increment) in a burst which sort of overwhelms the server at high send rate.**


Figure 5: Variation of burst size with time

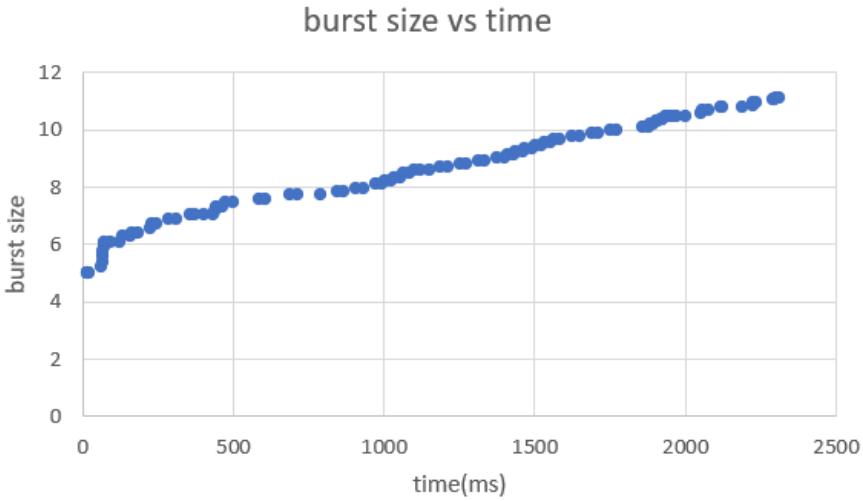


Figure 6: Zoomed-in view of Variation of burst size with time for first 2500ms

The plot above shows the variation of burst size with time based on replies, skipped requests, and squishes from server. It is initialised to 5. It always increases in a no squish scenario.

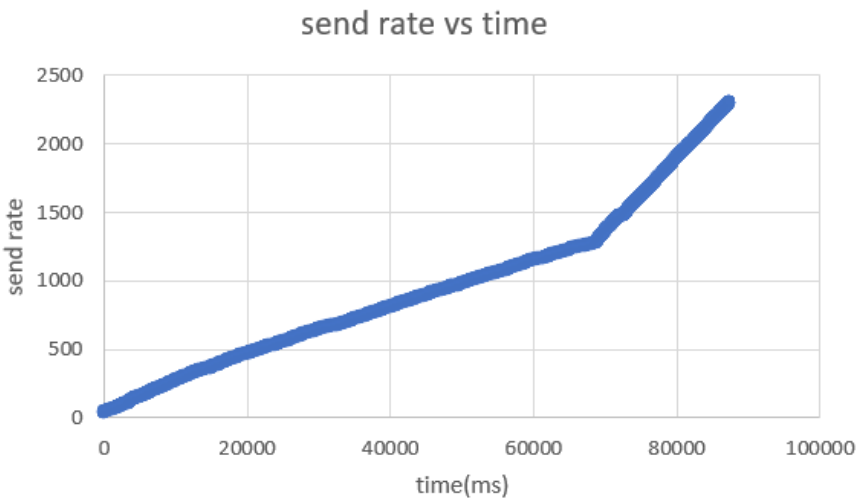


Figure 7: Variation of send-rate with time, based on replies from server

**Send-rate** is a variable in our client program which **decides the wait time**. It is initialised to be 40. We introduced a wait time of (1/send-rate) b/w each send request. Based on the squished packets, the send-rate is adjusted. If the send-rate decreases, that means we have been squished which is not seen here since have tried to implement in such a way to avoid squishing. Hence otherwise, the send-rate always increases.

Analysis on text file provided to test our code by variable rate vayu server hosted on port 9802:
After trying for various combinations of send rate,increment and burst,we arrived at a combination we could get decent time without squishing. No. of lines = 50500(approx)
No. of requests sent by main thread = 2266(approx)
No. of replies received by receiver thread = 2266(approx)
Time taken = **40-50 s**