Milestone 2

Kartik Meena , Priyadarshini Radhakrishnan

1 Code Working

- **What happens if a packet gets reordered on the server and you get replies for your requests in random order?** The server randomly decides not to reply every now and then. It uses a uniform random distribution to sample and provides a constant packet loss rate, so we can get different orders for our request resulting in storing the wrong data.
  We run two parallel threads one for sending the requests and the second one for receiving the responses. We made two lists, one to store the data of a particular offset and the second to check if a request corresponding to an offset is skipped.
  The send_request function runs in the main thread and requests data of 1448 bytes with some offset and the receive_messages function runs in a parallel thread in which we use the offset as an index to place the obtained data in the data list. Once a response is received, the value in the pending list at the corresponding index is changed to false. The index is obtained from the offset in the response message.
  The requests are sent in bursts and the size of the bursts is changed according to Additive-Increase/Multiplicative-Decrease logic. If replies have been received for all the requests sent then the burst size is incremented by 2 else the burst size is halved.
  In this way, all packet replies are positioned at the correct index and requests are sent only for skipped requests.

- **How are you sending your requests to not get squished and also maintain an optimal request sending rate ?**
  To do this we are sending requests in bursts of a certain burst size. The burst sizes are maintained using the AIMD approach.
  We have a list named **requests** which stores the indices that are to be requested for a burst and a variable which counts the number of received requests for a burst. If it is equal to the burst size, that means all the requests are answered and now we can successfully increase our burst size (here we increment the burst size by 2). Otherwise, if it is less than burst size that means some of the requests went unanswered and we half the burst size. This is done because, if some of the requests went unanswered it indicates a possible congestion of the network due to which packets might be dropped. We check if all requests in the previous burst have

been received or if some were declared lost. We then push out the next burst based on the AIMD-updated burst size and the **requests** list is updated with the requests to be sent for that burst.
We estimate the RTT and Timeouts using Exponentially Weighted Moving Average (EWMA), similar to what TCP does: Upon each reply that is received, the RTT estimate is updated using the value from this new sample. We give a sleep time of timeout after sending each burst of requests. So the rate at which we would be sending is,

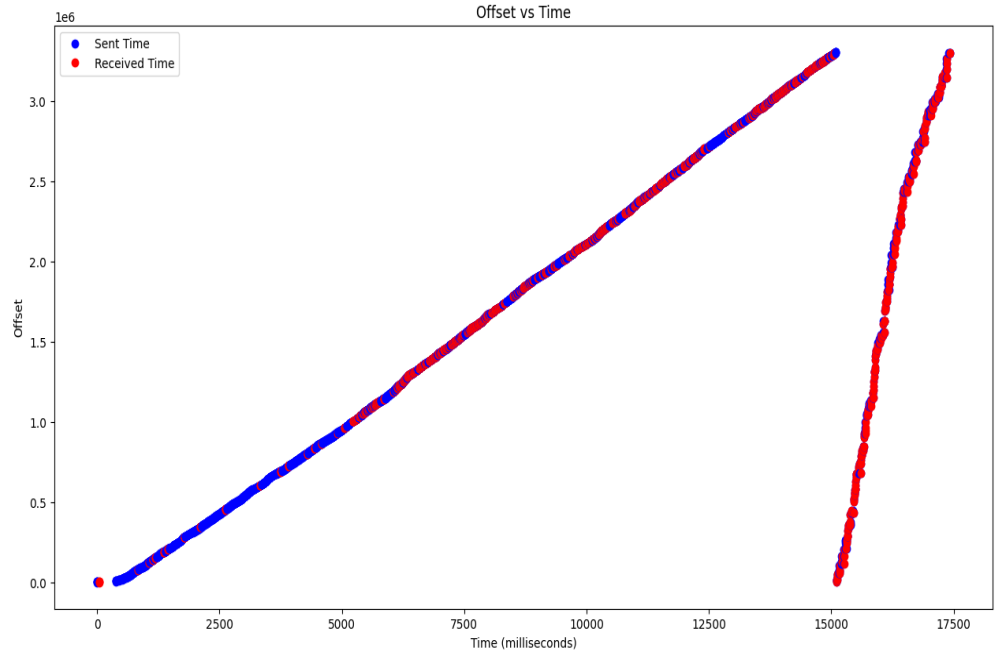
$$\text{Sending Rate} = \frac{\text{burst\_size} \times 1448}{\text{Timeout}}$$

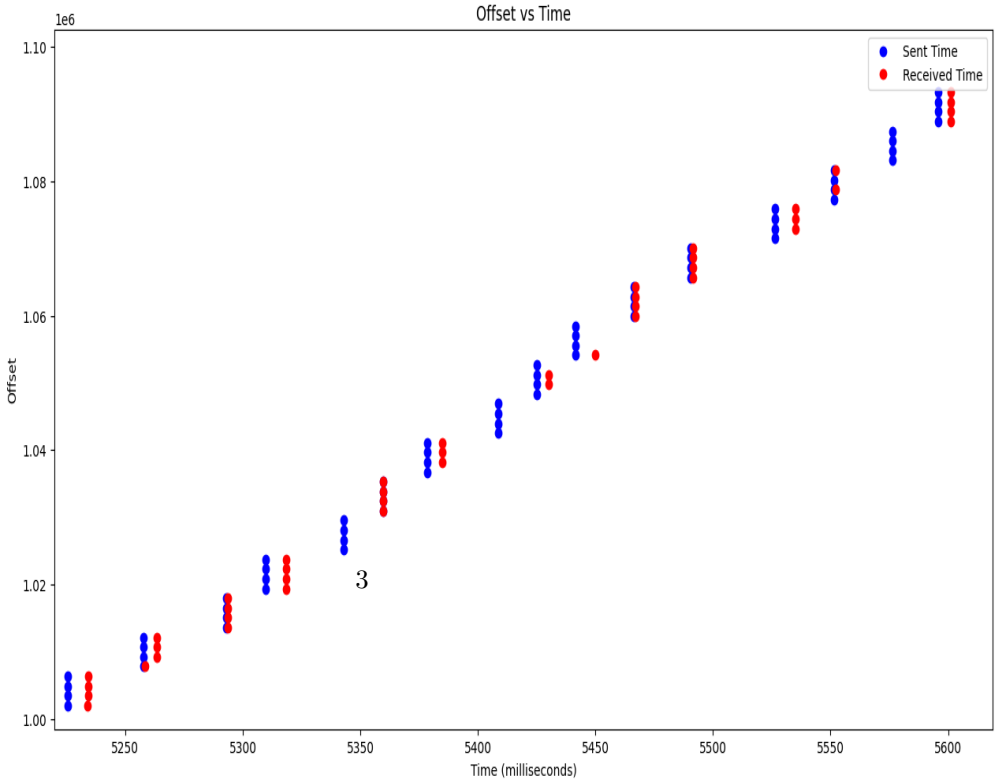
This rate is dynamic as the burst size keeps on changing. Since we have dynamic timeouts depending upon the estimated RTT from the server, our sending rate will be optimal and the client will never get squished. If the client gets squished in spite of all this, the client is let to sleep for a few seconds before it again starts sending requests to the server.

- **Conclusions:**
  - We experimented with both an AIMD client and a constant congestion window client.
  - For the constant burst size approach we don't change the burst size keeping it the same as the initial value. We chose our congestion window which is in this case equal to the burst-size*1448 bytes.
  - For a constant rate server we found out that the performance of a constant burst size client is better than an AIMD client. The time taken to address all requests for a constant client was around 15-17 seconds with a penalty in single digits (generally 2-8). Whereas for an AIMD client, the time taken to address all queries was 25-32 seconds with a penalty in single digits (generally 2-8).
  - Our client sends requests at a rate such that it might never get squished. But even then if it gets squished, then the burst size is halved and requests are sent at a constant rate since then, till it becomes un-squished. Once it gets un-squished, it again acts like a normal AIMD client with changing burst sizes, therefore with changing rate.

2 Graphs

2.1 Graphs for Vayu Server using constant burst size





```
MD5 hash for data: 103498de7c685fa2ee0b1b8a4926a9a7
Result: true
Time: 16821
Penalty: 5
```

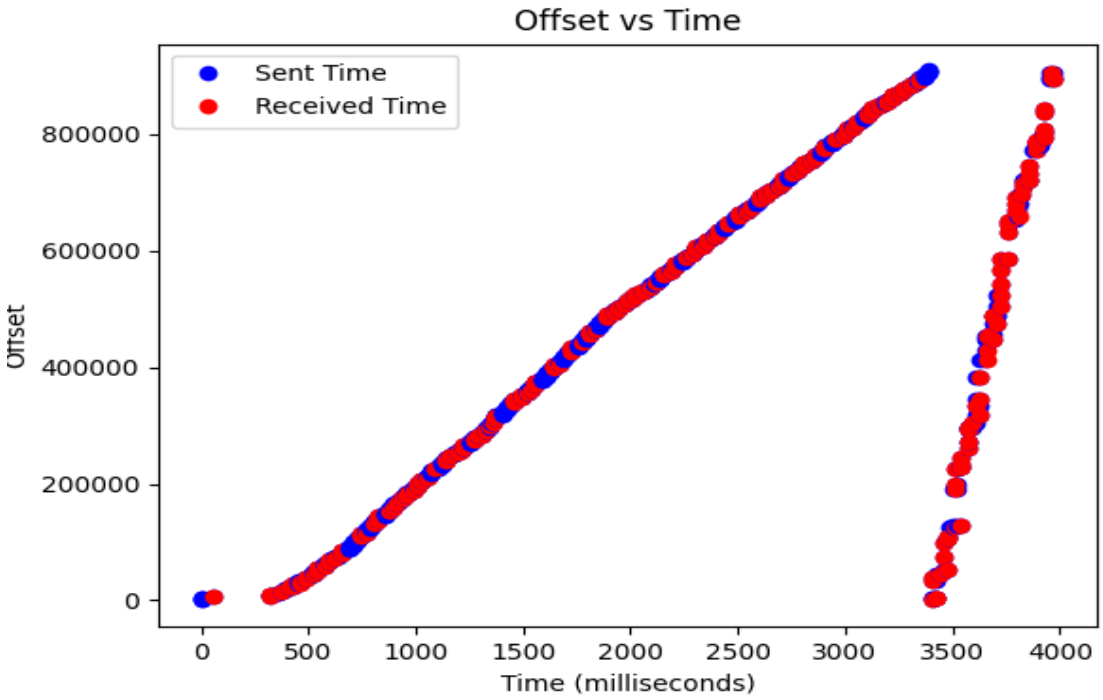
- **Figure 1**:
  It is visible in the graph as the first pass from 0 until approximately 14000 ms, we sent requests to the server only for the skipped requests, and finally, from 14000 ms to 17000 ms, we got the reply for all those requests which were skipped the previous time. Requests are shown in blue and replies are in red.
  Here this time we are taking 2 rounds as we are sending only some fixed burst sizes, so the possibility of getting skipped requests is reduced.

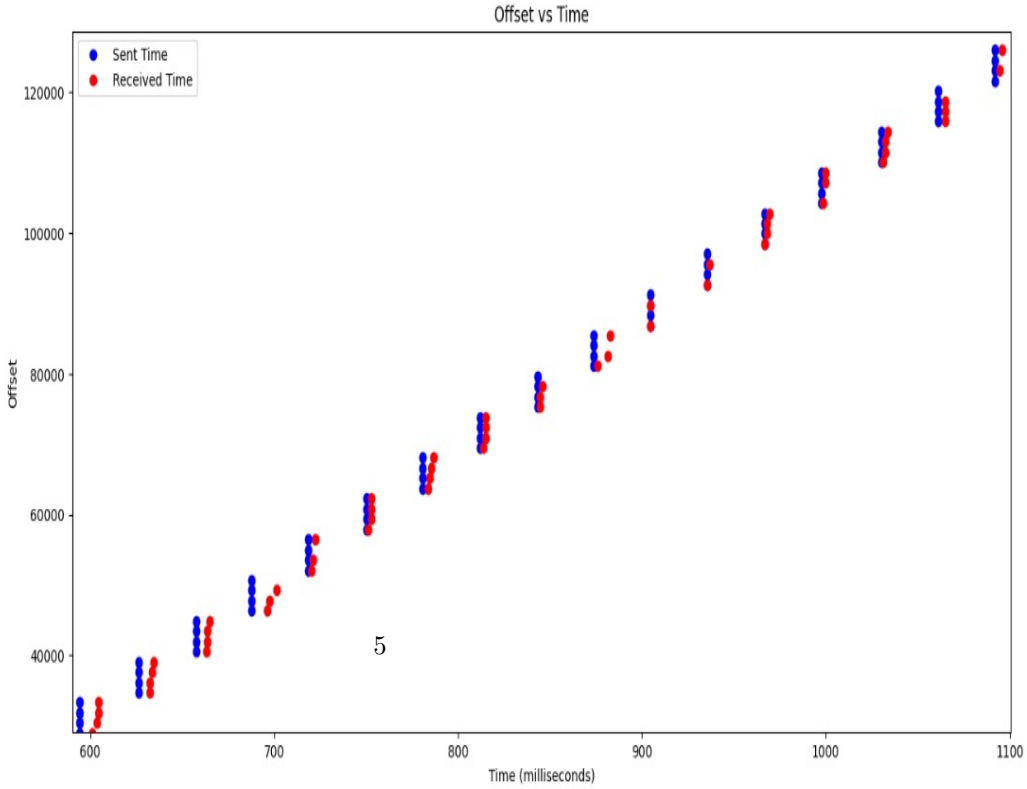
- **Figure 2** : Figure 2 shows a zoomed-in view of this trace for 5250 to 5660. we are sending a burst size of 4 each time, we can see that some of the requests are received immediately which overlap with each other or with slight delays (have some gap between reply and request) or a few skipped requests (number of blue dots is not equal to red dots). This spacing starts with a large conservative value and is successively reduced to approach an ideal gap wherein almost all data from the previous burst has been received and now a new burst can be issued.

- **Range of time and Penalties :** We are getting a time range of 16-27 sec and penalties between 0-30 only, not getting squished anywhere.

- **Curious Points :** We can see in Figure 1 and also in Figure 2 that a lot of requests are skipped or suffer some delay this suggests that the Vayu server is moderately congested and has high traffic which is indeed true as lot of people are trying to receive data form vayu. In Figure 2, there is also some delay between the consecutive requests also, which suggests that we were going fast and we had given some sleep time equal to timeout to check if packet is lost or not .
  Constant burst sizes can reduce the overhead associated with dynamically adjusting burst sizes or congestion control mechanisms. This simplifies the program and reduces computational requirements. By maintaining a fixed burst size, the protocol can prevent overloading the network with sudden spikes in data transmission, which may lead to congestion and packet loss.

2.2 Graphs for localhost using Constant burst size





```
MD5 hash for data: 4b2f35de5fe0c7fa4caf762cc848ca7c
Result: true
Time: 3981
Penalty: 3
```

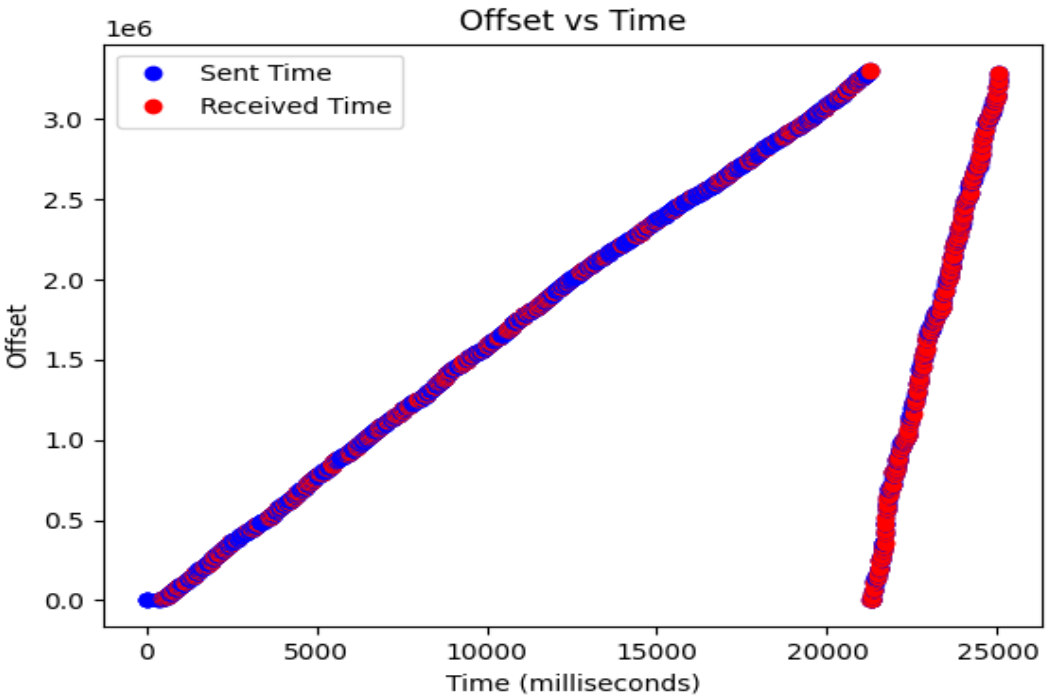
- **Figure 1**: It is visible in the graph as the first pass from 0 until approximately 3000 ms, we sent requests to the server only for the skipped requests, and finally, from 3000 ms to 4000 ms, we got the reply for all those requests that were skipped the previous time. Here this time we are taking 2 rounds as we are sending only some fixed burst sizes, so the possibility of getting skipped requests is reduced.

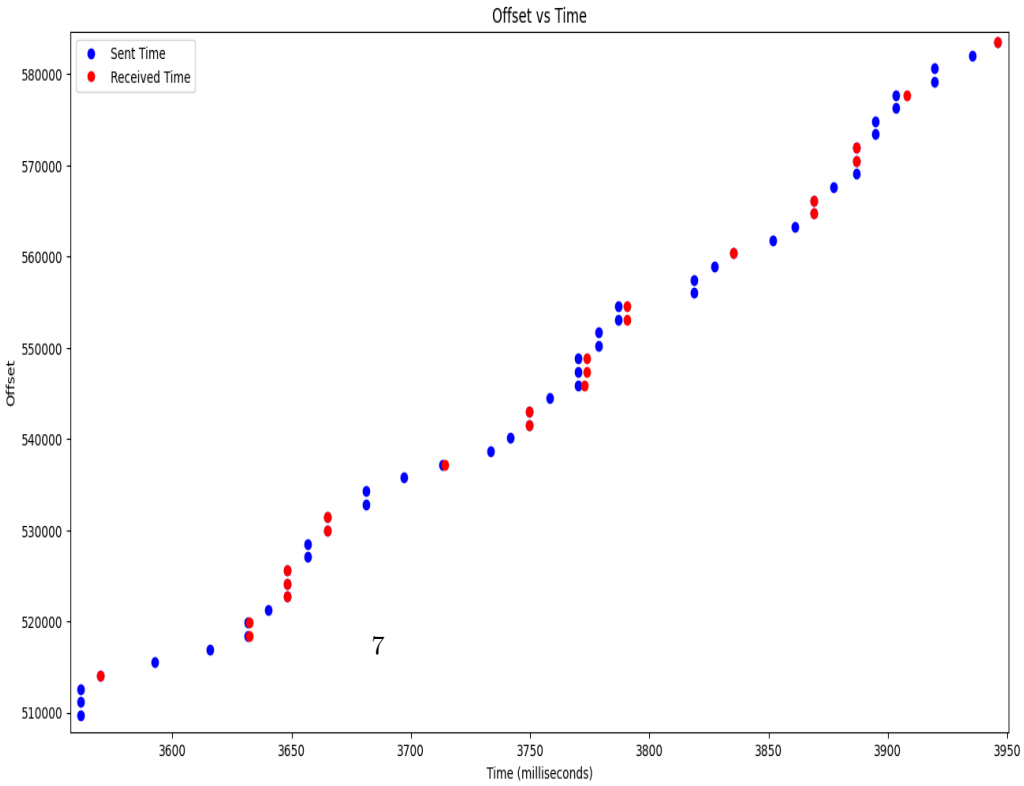
- **Figure 2** : Figure 2 shows a zoomed-in view of this trace from 600 ms to 1100 ms. we are sending a burst size of 4 each time, we can see that some of the requests are received immediately which overlap with each other or with slight delays (have some gap between reply and request) or a few skipped requests (number of blue dots is not equal to red dots). This spacing starts with a large conservative value and is successively reduced to approach an ideal gap wherein almost all data from the previous burst has been received and now a new burst can be issued.

- **range of time and Penalties :** we are taking time around 3.5 - 8 sec and penalties in the range 0-20 without getting squished at any time.

- **Curious Points :** For localhost we can see that most request are received quickly without much delay and only a few requests are getting skipped, this gives us the idea that our local server is not congested and have low traffic which is indeed true as there is nobody else is receiving data from our host. In Figure 2 we can see the gap between consecutive requests is more or less the same suggesting that we are not requesting too fast.The constant burst size behavior is deterministic and doesn't depend much on external factors like network conditions or congestion. By maintaining a fixed burst size, the protocol can prevent overloading the network with sudden spikes in data transmission, which may lead to congestion and packet loss.

2.3 Graphs for Vayu server using AIMD

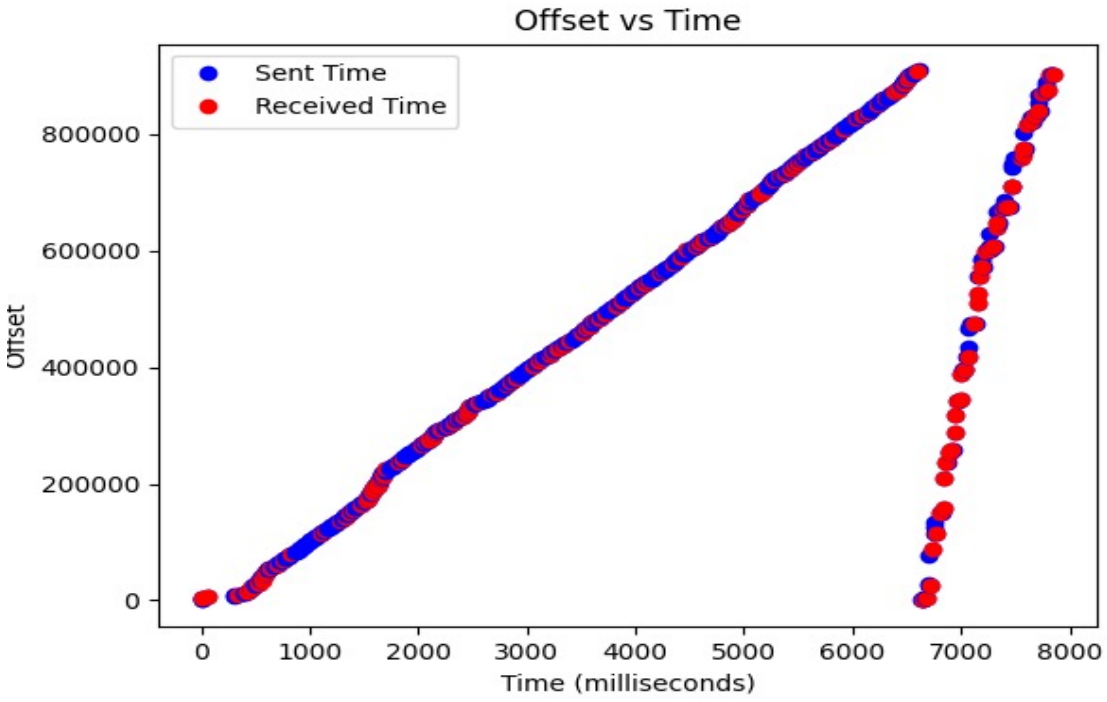


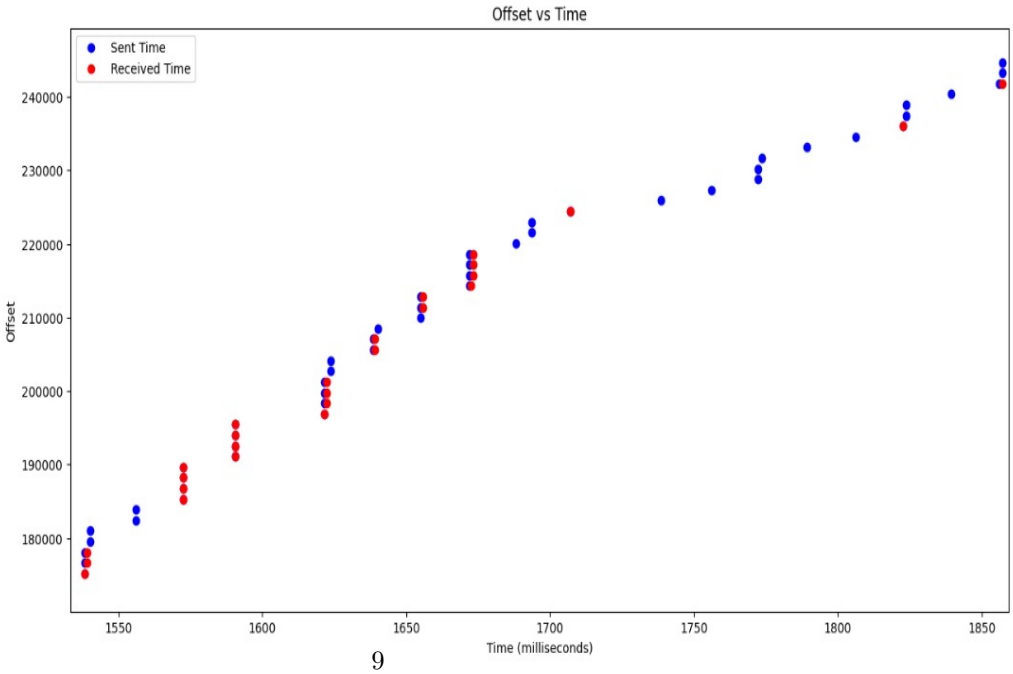


```
MD5 hash for data: 103498de7c685fa2ee0b1b8a4926a9a7
Result: true
Time: 25072
Penalty: 3
```

- **figure 1** It is visible in the graph as the first pass from 0 until approximately 20000 ms, we sent requests to the server only for the skipped requests, and finally, from 20000 ms to 25000 ms, we got the reply for all those requests that were skipped the previous time.
- **figure 2**: in the graph it can be seen that the burst size of the request is not constant, as whenever we don't get a reply we decrease the burst size. The is not much overlapping between requests suggesting that there are a lot of replies that are out of order and missed and our send function uses some sleep time to get this order correct and the missing requests.
- **range of time and penalties :** we are taking around 25-38 sec to get the complete file and penalties are around 0-30.
- **Curiours Points :** We can see that this approach take more time and penalties then constant burst size here are some of the reasons:
  1) sometimes causes self-inflicted congestion, where a flow reduces its sending rate when the network could potentially handle more traffic, leading to underutilization.
  2) AIMD reacts to congestion by reducing its congestion window multiplicatively, which can be slow to recover from congestion. It can take many round-trip times to reach the optimal congestion window size, leading to suboptimal throughput during the recovery phase.
  3) One of the reasons of Graph is non-linear is the incrementing factor is dependent on the previous rate which changes with each request.

2.4 Graphs for localhost using AIMD



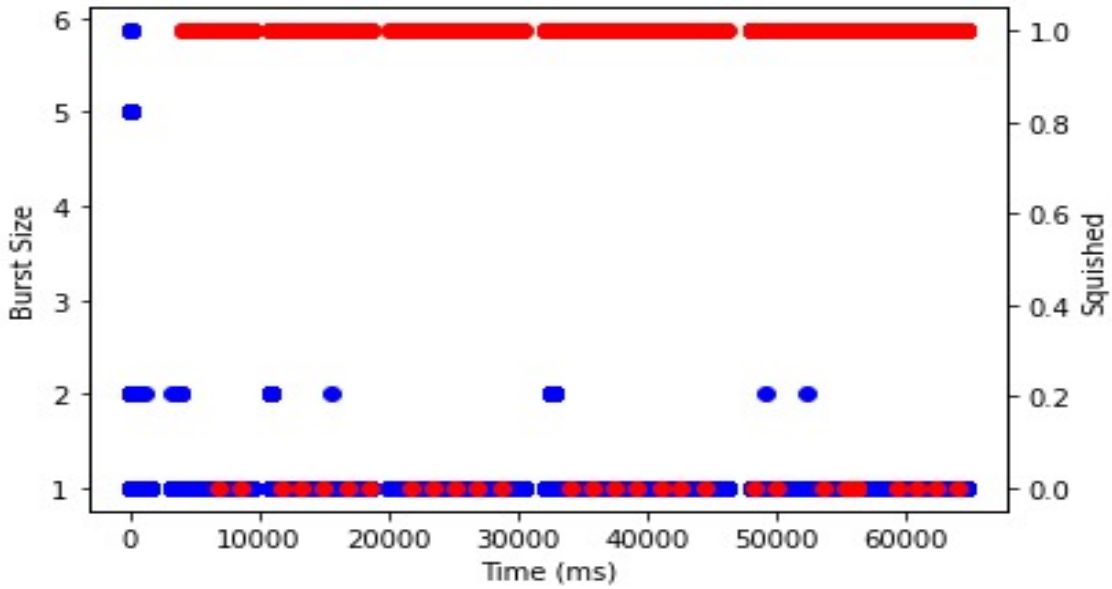


```
MD5 hash for data: dc907ea3f293707a097940a77af3f565
Result: true
Time: 7354
Penalty: 3
```

- **range of time and penalties :** we are taking around 5 - 8 sec to get the complete file and penalties are around 0-20.
- **Figure -1** It is visible in the graph as the first pass from 0 until approximately 6000 ms, we sent requests to the server only for the skipped requests, and finally, from 6000 ms to 7000 ms, we got the reply for all those requests that were skipped the previous time.

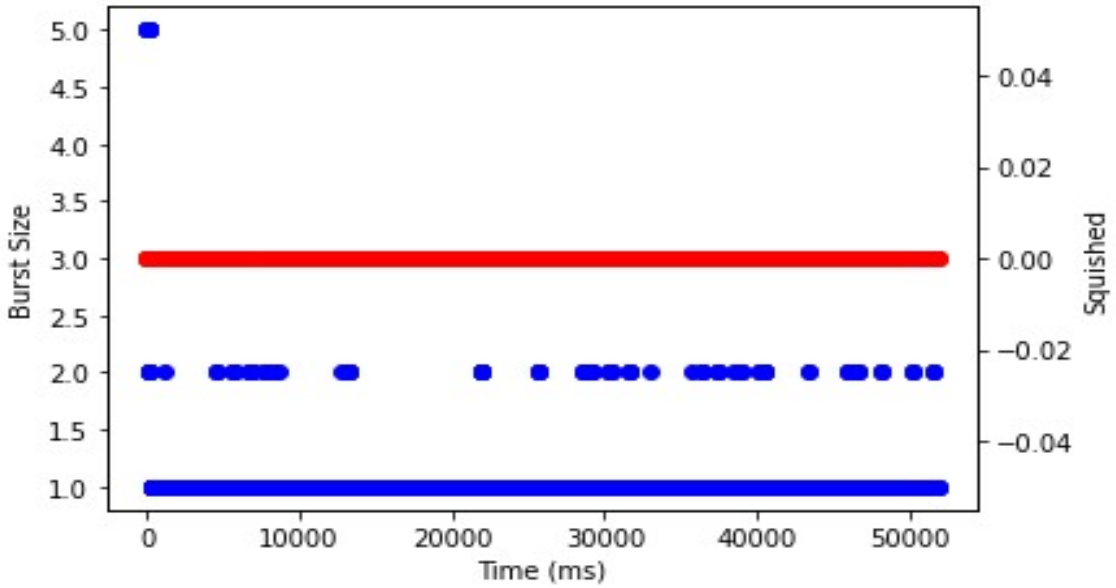
2.5 Burst-size for an AIMD behaving client

- For a squished client:



Here the client gets continuously squished. Whenever it gets squished its rate is reduced to half of the initial value and remains constant until it gets un-squished.

- For a non-squished client:



Here the client never gets squished. The requests are sent at a rate such that it would never get squished, minimise the penalty and as well as time. This is done by estimating the RTT properly.

2.6 Burst-size for a constant burst-size client

- For a squished client: