COL334 Assignment-3 Report

Amaiya Singhal 2021CS50598
Om Dehlan 2021CS10076

Milestone 2

Approach and Methodology

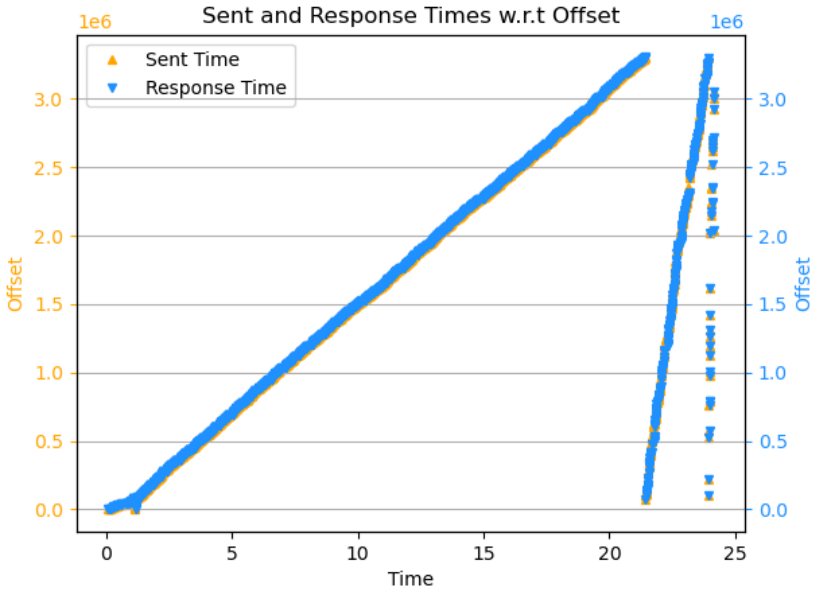
- We have implemented the **AIMD (Additive Increase Multiplicative Decrease)** policy for congestion control.
- We have used 2 threads for this, one thread is for sending the requests and another to simultaneously receive the responses from the Vayu server.
- Some modifications were made to the general AIMD policy to find the best rate for a **Fixed** Rate Leaky Bucket Server, to give the best time while minimising the penalty.
- The specifics of our approach are described below:
- **Sending and Receiving Packets:**
  - There are 2 separate threads for sending and receiving.
  - On one thread, requests are send sequentially to the server, incrementing the offset each time by 1448 bytes (Packet Size).
  - On the other thread, the responses are being received from the server and the data received is stored in a dictionary with the offset as the key.
  - Requests are sent until all the packets are received, i.e. the size of the dictionary equals the expected number of packets.
- **Variation in Burst Size:**
  - The burst size is decreased to half of its value when there is more than or equal to **20% drop** in the packets sent.
  - This is done because sometimes a packet is dropped but it does not mean that the rate is too high, hence the 20% value is used as tolerance.
  - Otherwise if the loss is less than 20% the burst size is incremented by 1.
- **Gap between the Bursts:**
  - First 50 requests are sent in bursts of 5. The corresponding time taken to receive each of the bursts is used to calculate a running average of the **RTT**.
  - This is done so as to get a good estimate of the effective RTT from the server.
  - This value of RTT is used for sending the further bursts with the AIMD policy.
- **Ensuring Correctness:**
  - A dictionary is used to store all the data received and requests are sent to the server until the size of the dictionary equals the number of packets of size 1448 required.
  - This ensures that requests are sent until there exists an entry for each of the offset values in the dictionary.
  - Then the MD5 hash is calculated for the entire data and submitted to the server for verification.

Observations

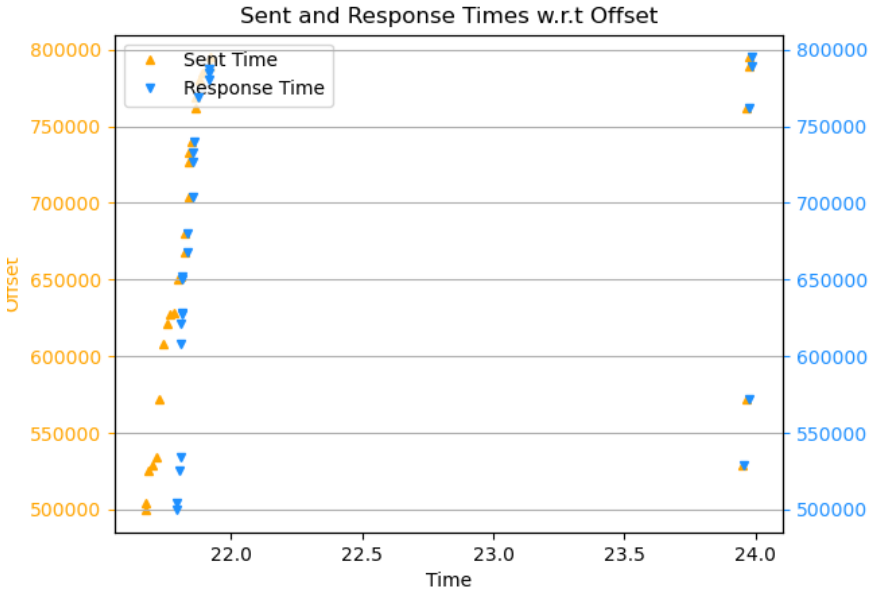
Run Number	Time (s)	Penalty
1	19.726	40
2	21.777	5
3	24.208	1

Time and Penalty with Run Numbers

- We are getting a time of around 15-25 seconds on the Vayu Server with this policy.
- The variation in time and penalty can also be explained by our implementation.
- We calculate the RTT as the running average of the times taken for the first 50 requests in bursts of 5.
- There can be variations in this time if too many or too less packets get dropped.
- If the RTT comes a little low then the total time reduces as the time between bursts is lesser however this leads to increased penalty.
- On the other hand, a higher RTT increases the time but leads to a much lower penalty.
- The graphs below explain our approach in more detail.

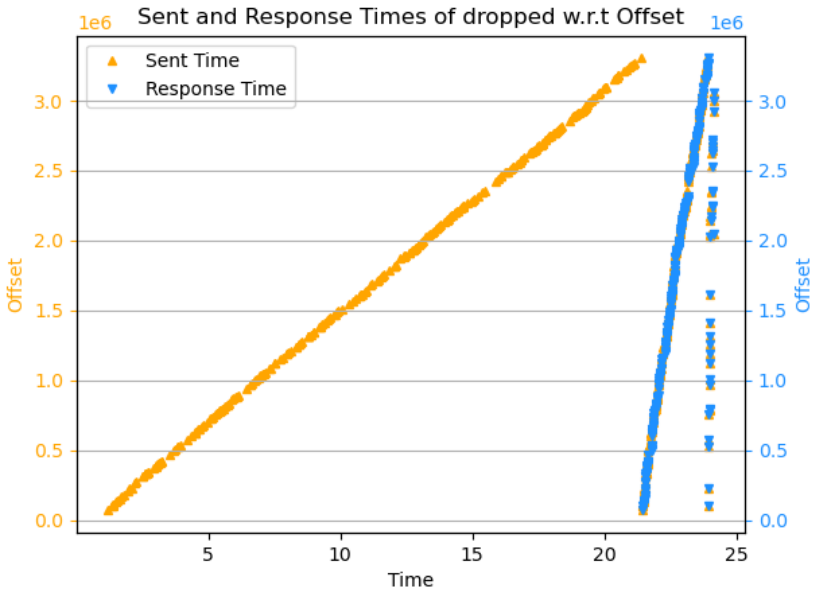


- The above graph shows the **Sequence Number Trace** for the requests sent and the responses received for the AIMD approach.
- All the requests are monotonically sent (in orange) and the responses are received for most of them (in blue). Due to the scale of this graph, it is not very clearly which responses are not received.
- Then the requests are sent again for the offsets for which we did not receive a response.
- This process had to be repeated until all the data was received and the MD5 hash could be submitted for verification.

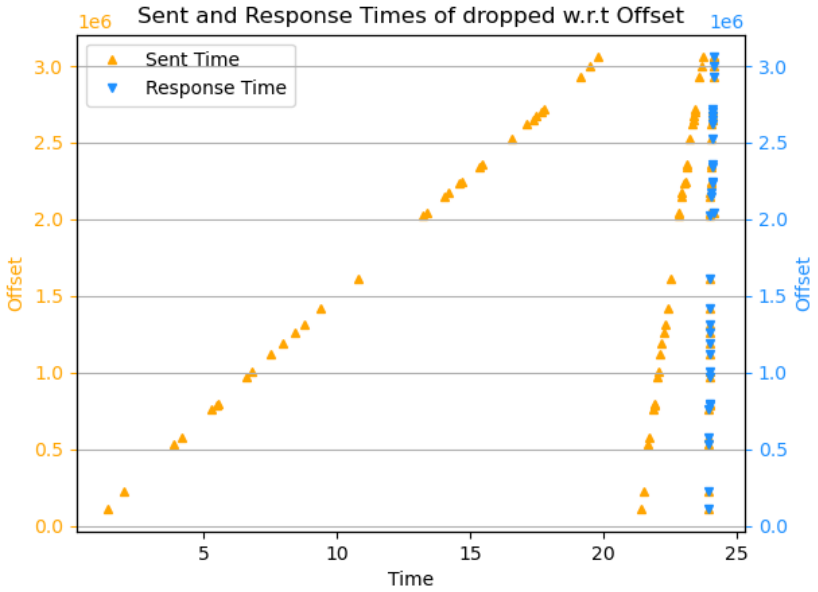


- The above graph shows a zoomed in version of the Sequence Number Trace.
- Here we can see that for some of the requests a response was not received (Orange triangle present but no corresponding Blue triangle indicating that a response was not received).
- Even in the second iteration some of the packets were dropped and further iterations were needed until all the remaining data was finally received.
- The requests sent in bursts can be seen together in the graph, and the gap between the groups represents the gap between the bursts.

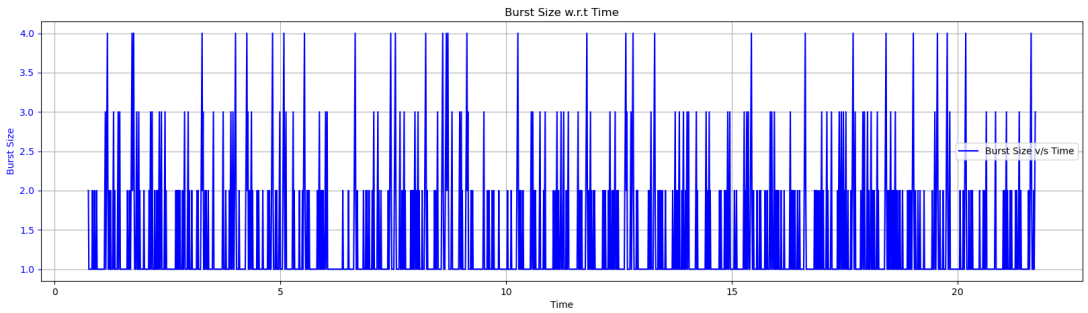
Some More Graphs

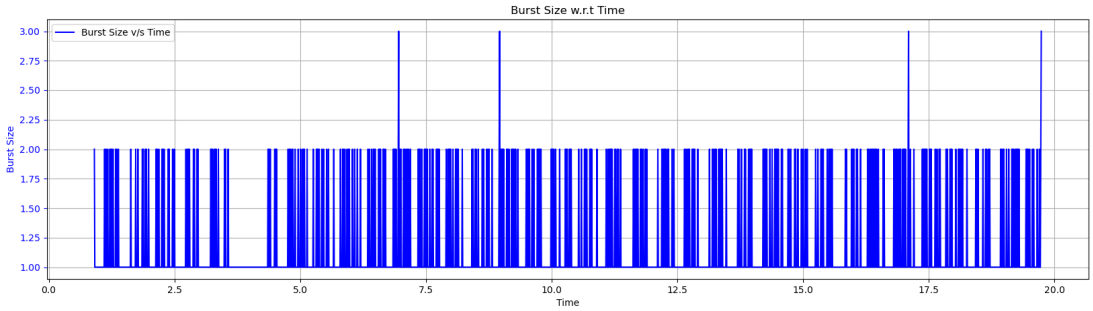


- The first graph shows all the packets that were dropped in the first run. (There is no blue triangle for any of them indicating that no response was received).
- Therefore further iterations are called for these.



- The second graph shows all the packets that were dropped in both the first and second runs. (There is no blue triangle for any of them indicating that no response was received).
- Therefore further iterations are called for these.





Burst Size v/s Time

- The above graphs show the **Change in Burst Size of requests w.r.t Time** for our approach.
- We can observe that the graphs goes up and down representing the AIMD approach.
- Notice the variation in the burst sizes among the 3 runs, this is due to the different value of RTT set in them.
- A higher RTT means that the burst size is increased and a lower RTT means that the burst size gets decreased. This happens in a manner that the effective rate keeps near the optimal rate for the server.

Alternate Approach

- We have also implemented a **Clever Hack Sequential** policy for congestion control.
- We have used 2 threads for this, one thread for sending the requests and another to receive responses from the Vayu server.
- The sending thread sequentially sends requests to vayu with a variable rate, i.e., if we are receiving responses for a large ratio of requests, we increase the rate and if we receiving less responses for the requests, we decrease the rate.
- We check the ratio of responses received for every 20 requests and we multiply the rate by the proportion of packets received to the requests sent i.e. 20. A constant factor is added to the numerator to ensure that the rate can be increased too and is not only a decreasing function.
- There is also a maximum bound set whenever a specific rate leads to a lot of drops (more than 25%). This ensures that the rate does not increase to the same value again.
- This is a clever hack because for a constant rate server, our approach quickly converges to the optimal rate of requests giving the best results overall.

Observations

Run Number	Time (s)	Penalty
1	13.073	48
2	12.592	18
3	13.918	35

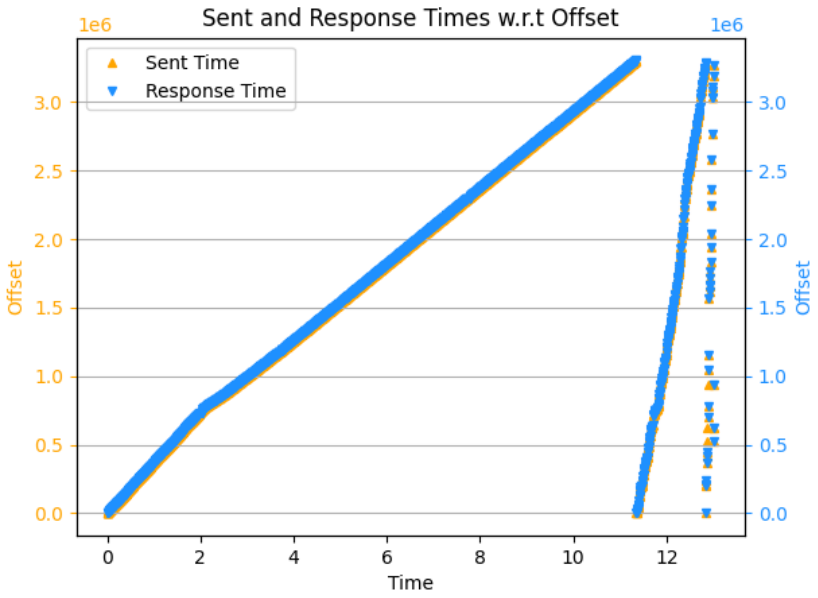
Time and Penalty with Run Numbers

- We can see from the results that this approach works better than the AIMD client.

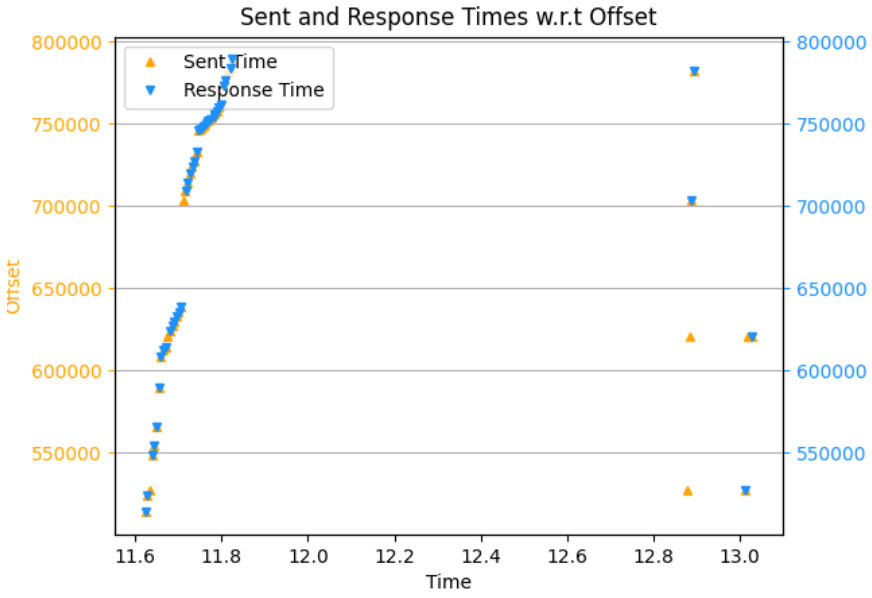
- This is mainly because instead of a multiplicative decrease like in AIMD, here the rate is constantly being adjusted to find an optimal rate and the requests are sent at that rate.
- Even if there are more losses (indicated by the higher penalty), since there are no squishes packets are continued to be sent at a higher rate, unlike halving the burst size like in AIMD.
- Hence this approach is faster, however leads to a slightly higher penalty.

Graphs

Sequence Number Trace

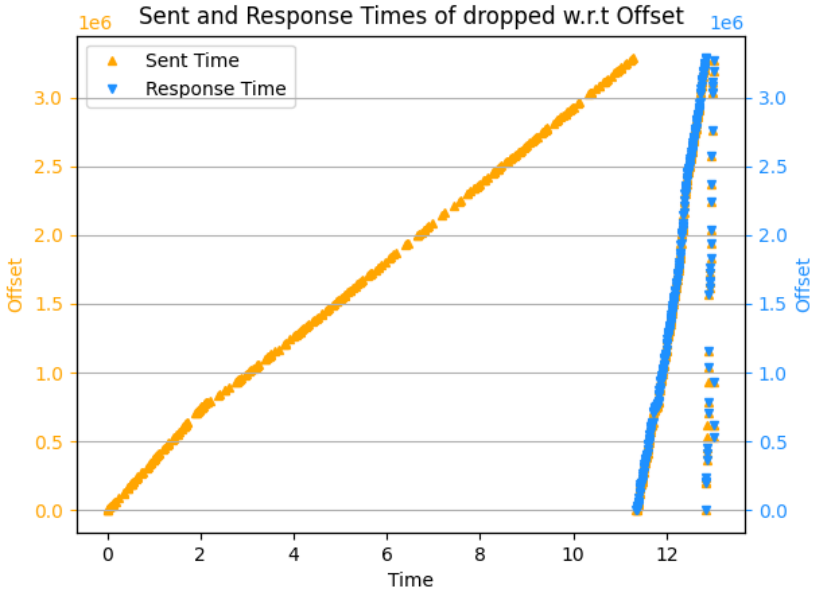


- The above graph shows the **Sequence Number Trace** for the requests sent and the responses received.
- All the requests are monotonically sent (in orange) and the responses are received for most of them (in blue). Due to the scale of this graph, it is not very clearly which responses are not received.
- Note the slight change in slope of the graph showing a change in the rate of sending requests.
- Then the requests are sent again for the offsets for which we did not receive a response.
- This process had to be repeated until all the data was received and the MD5 hash could be submitted for verification.



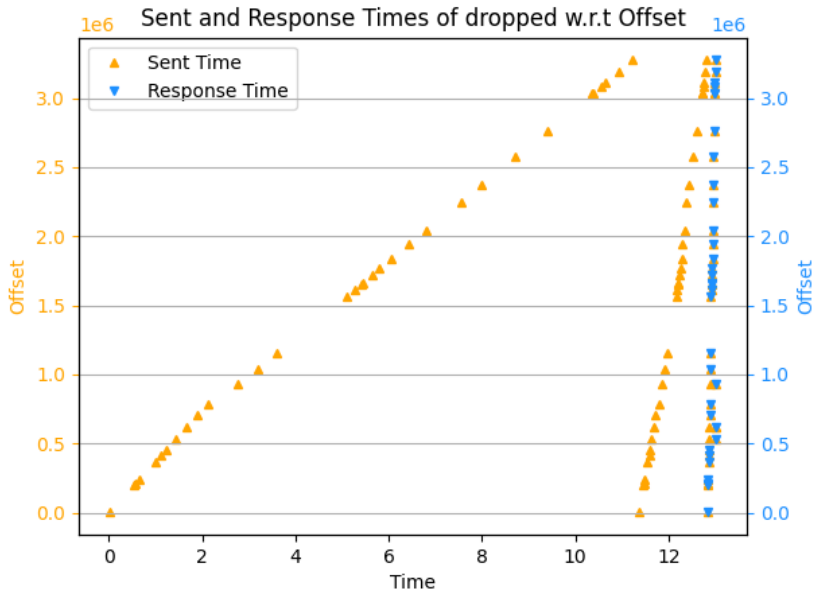
- The above graph shows a zoomed in version of the Sequence Number Trace.
- Here we can see that for some of the requests a response was not received (Orange triangle present but no corresponding Blue triangle indicating that a response was not received).
- Even in the second iteration some of the packets were dropped and further iterations were needed until all the remaining data was finally received.

Some More Graphs

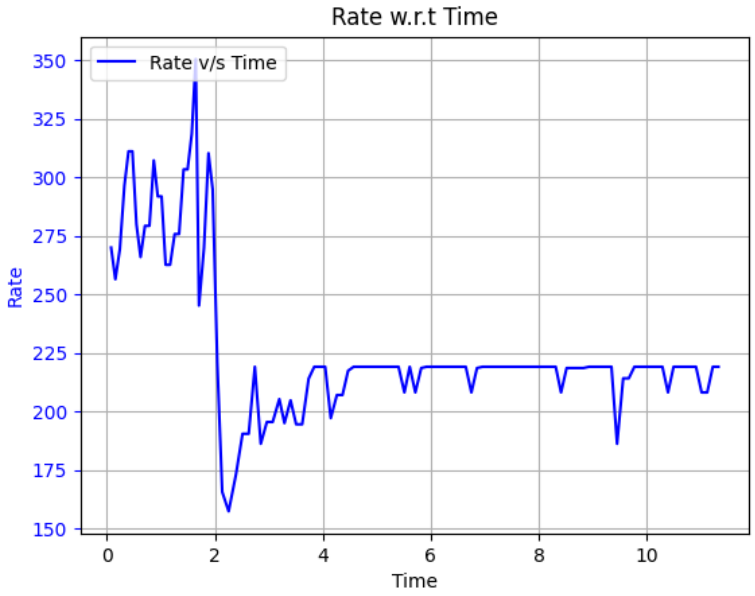


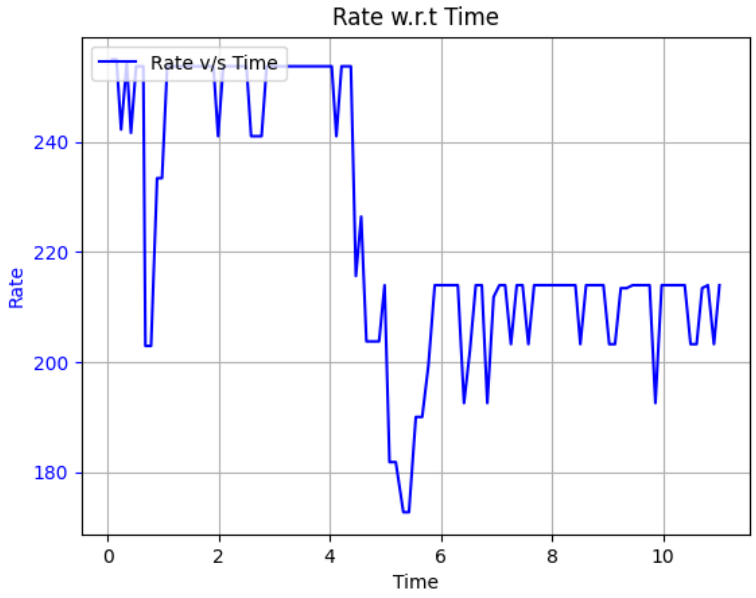
- The first graph shows all the packets that were dropped in the first run. (There is no blue triangle for any of them indicating that no response was received).

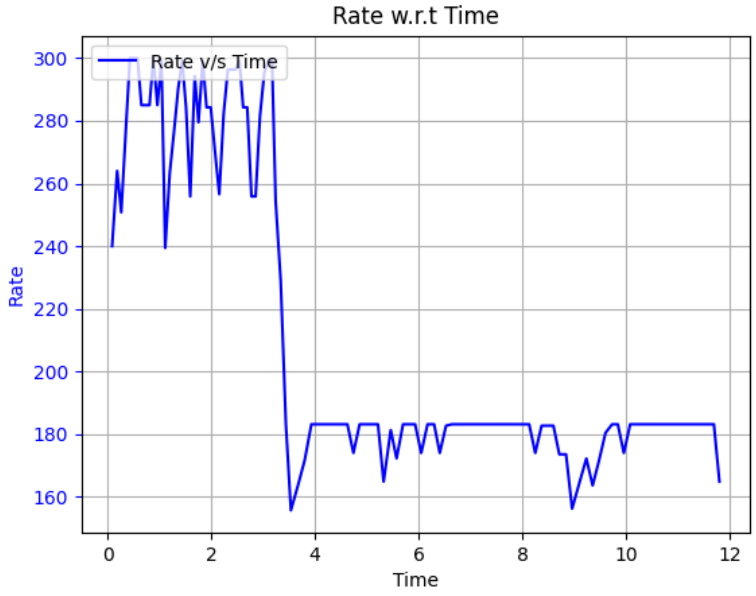
- Therefore further iterations are called for these.



- The second graph shows all the packets that were dropped in both the first and second runs. (There is no blue triangle for any of them indicating that no response was received).
- Therefore further iterations are called for these.







Rate v/s Time

- The above graphs show the **Change in Rate of requests w.r.t Time** for our approach.
- We can observe that the graphs goes up and down as our algorithm tries to find an optimal rate to send requests to the server.
- Notice that the rate converges near to a particular rate for longer duration than other rates, which should be our optimal rate.