COL334 Assignment 3

Mihir Kaskhedikar 2021CS10551
Payas Khurana 2021CS50948

October 2023

1 Week 1

We have used **threading** to send requests parallelly to the server. Firstly we created a UDP socket and requested the number of bytes on it. After receving the `total_bytes`, we divide them almost equally in `num_threads` continous portions where `num_threads` were the number of threads we used. For each thread we created a new socket,ran a while looop and requested data from it such that the range of data requested (**[Offset,Offset+NumBytes)**) entirely lies in the portion of data assigned to this thread. NumBytes is assigned the value $\mathbf{min}(\mathbf{1448}, \mathbf{limit} - \mathbf{offset})$ where limit is the ending byte index of the data assigned to this thread. Similarly Offset is assigned the value *start* (starting byte index of data assigned) in the beginning and once we correctly receive the socket response for this data request, it is incremented by NumBytes to form the Offset in the next iteration of the while loop. If we dont receive the data, we dont change the Offset value and simply continue. In order to ensure that the response received from the socket is correct, we performed 2 important checks:

- Checking the length of actual data received (that is the response received after removing the "Offset:" and "NumBytes:" lines) is equal to NumBytes
- Checking the value mentioned in the "Offset:" field of response matches with the Offset value that we requested the socket to send.

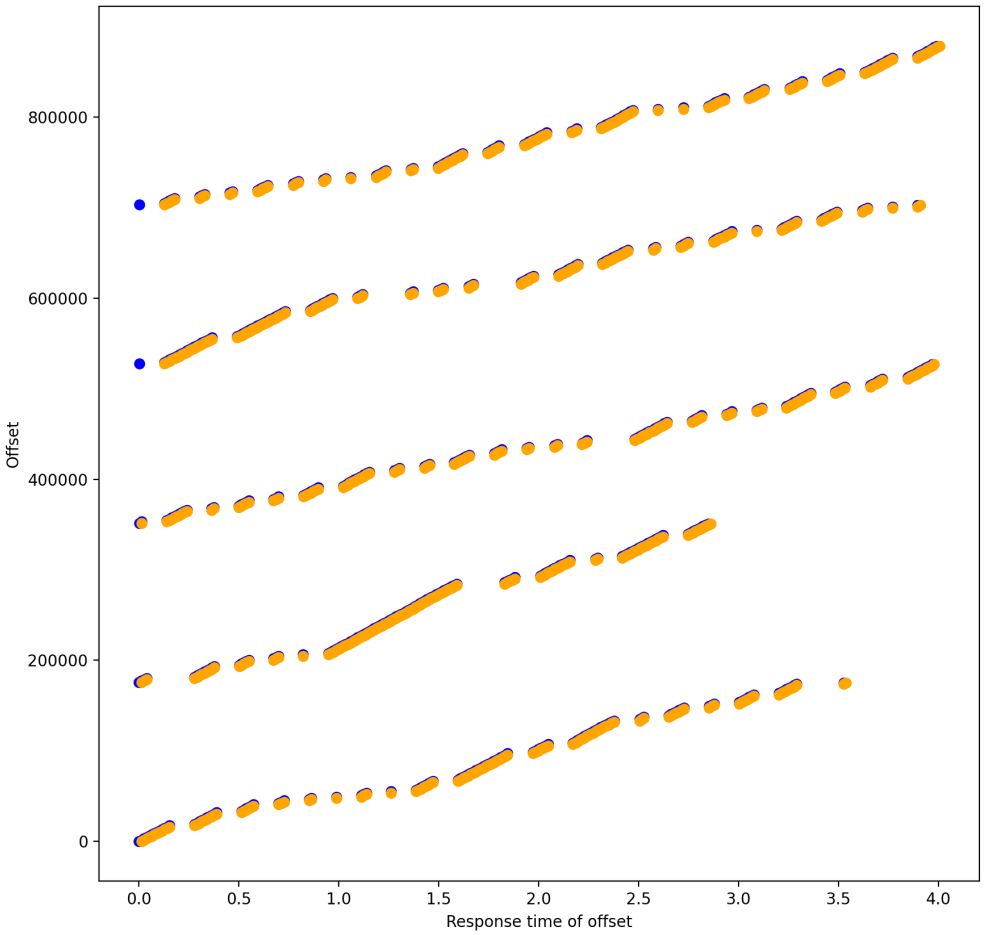
If the response is received correctly, we append it to a string corresponding to the thread and change the Offset value. We have implemented 2 features to avoid token loss:

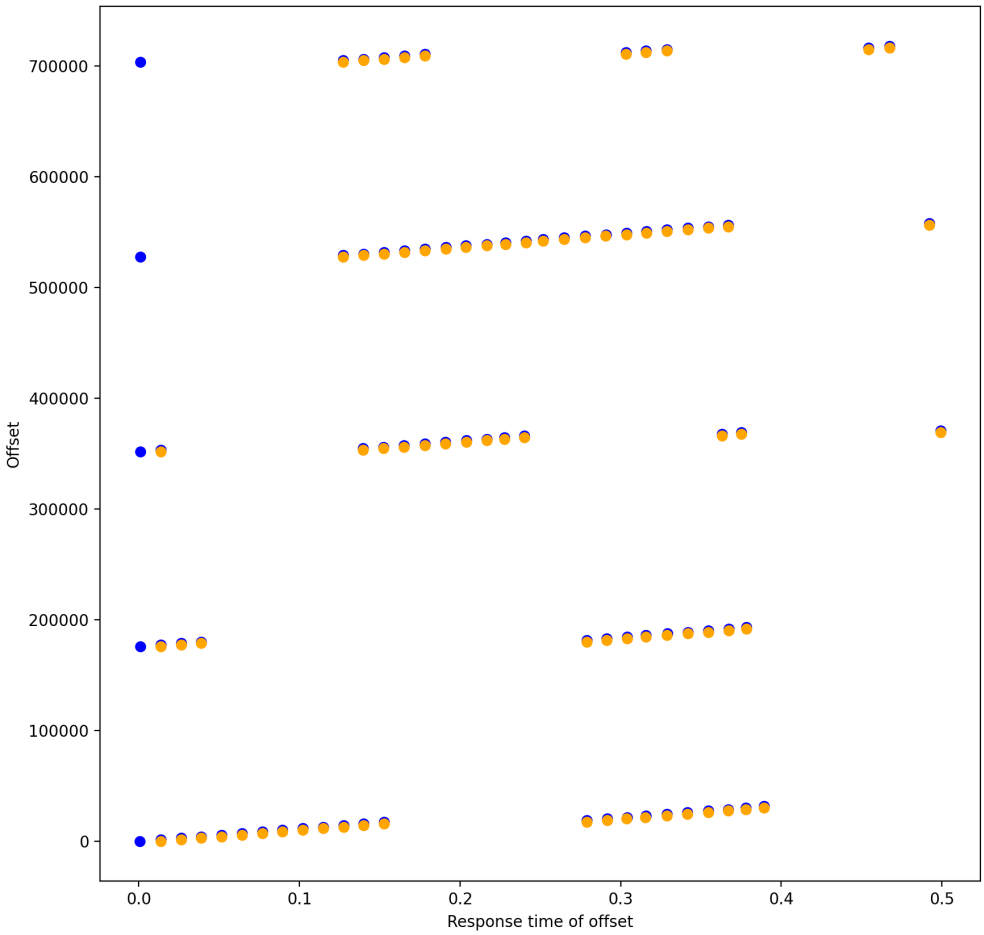
- In each thread, before we make a request to the socket we do **time.sleep(0.01)**.
- We use the **try-except** syntax to send or receive anything from the socket everytime with a cutoff of $0.1s$.

After completion of all threads we concatenate the strings of the threads and send it to the server. Below is the offset vs response time graph, when `num_threads` $= 5$. We were only able to test our code on local server (Vayu was not responding). The first image gives a complete picture of the request and reception times for all offsets (blue dots specify the request times and orange dots, the reception times, since they would be very close to each other for a particular offset and the offset after it, orange dots almost cover the blue ones). The second image gives a magnified image of the request and reception times between $0s$ and $0.1s$. From both the images one can clearly deduce that 5 threads were used for sending requests.

2 Plots for Week 1 on vayu server

We tried our client on the server hosted at `10.17.7.134` by keeping socket timeout as **0.05s** and **constant sleep time** of **0.01s** between two consecutive requests on a thread. It took **13.811s** with **8** penalties. The plots are shown in Figures 1 and 2.


800000
600000
400000
200000
0
Offset
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
Response time of offset


700000
600000
500000
400000
300000
200000
100000
0
Offset
0.0
0.1
0.2
0.3
0.4
0.5
Response time of offset

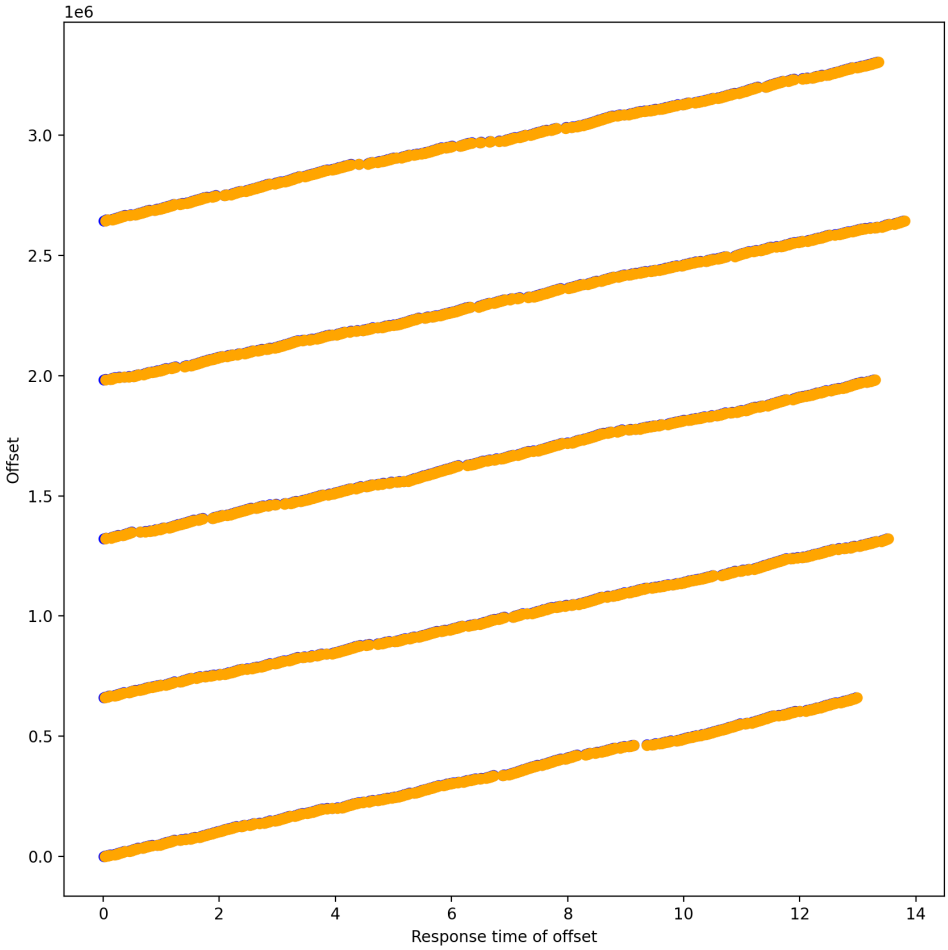


Figure 1: Offset vs response time for vayu server

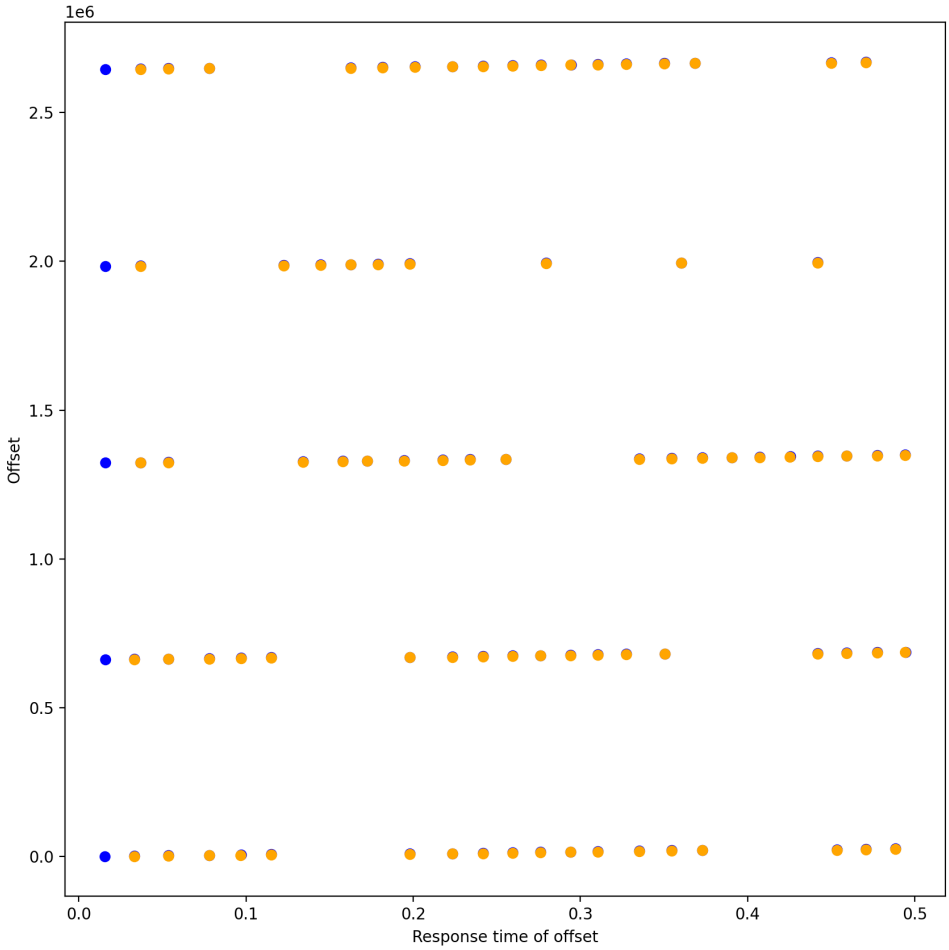


Figure 2: Magnified offset vs response time between 0-0.5s for vayu server

3 Week 2

We have used the **AIMD** strategy used by TCP to control congestion in the server. After experimenting with different values, we found that an initial request rate of 200 requests/s, Additive Increment of 5, and Multiplicative Decrement of 1.75 to give the best result i.e. a fair balance between time taken and number of penalties conceded.

So, continuing from milestone 1, we kept a rate at which we sent requests to the server, i.e. time between two requests is $1/rate$. Now, if we detect a packet loss or experience a bad request due to lack of tokens in the bucket, we divide the rate by 1.75. However, if the request is successful, we will increment the rate by 5. And since we are using multiple threads to cut down our losses, we will do the same independently for each thread (it might be that different threads are simultaneously operating at different rates).

So based on initial few requests, the client adjusts to the token generation rate of the server by following the strategy as described above, then it oscillates around the true value (i.e. hacky one), but due to time taken in the adjustment, and the variance in further oscillation, it doesn't perform as good as hacky solutions, but that's expected. However it can now adjust itself to different servers with different rates, and even variable rate ones. We are maintaining a fair balance between time and penalty conceded, and avoiding squishing completely. Although, by tweaking the parameters of AIMD, we can do both - reduce time with increase in penalty, or increase time with decrease in penalty.

AI	**MD**	**Time taken**	**Penalty**
5	1.75	15.535s	51
5	2	18.420s	16
5	1.6	14.564s	86

4 Plots for Week 2 on vayu server

We tried our client on the server hosted at `10.17.7.134` by keeping socket timeout as **0.01s**, **AI** = **5** and **MD** = **1.75**. It took **15.535s** with **51** penalties. The plots are shown in Figures 3 and 4.

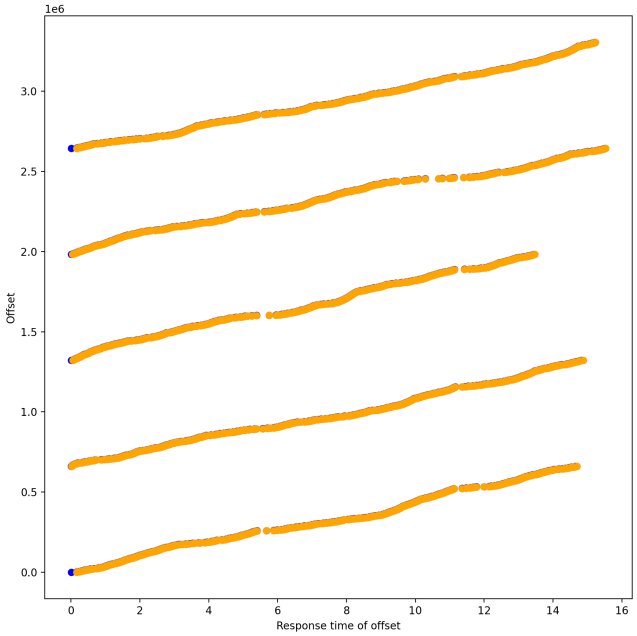


Figure 3: Offset vs response time for vayu server with AIMD

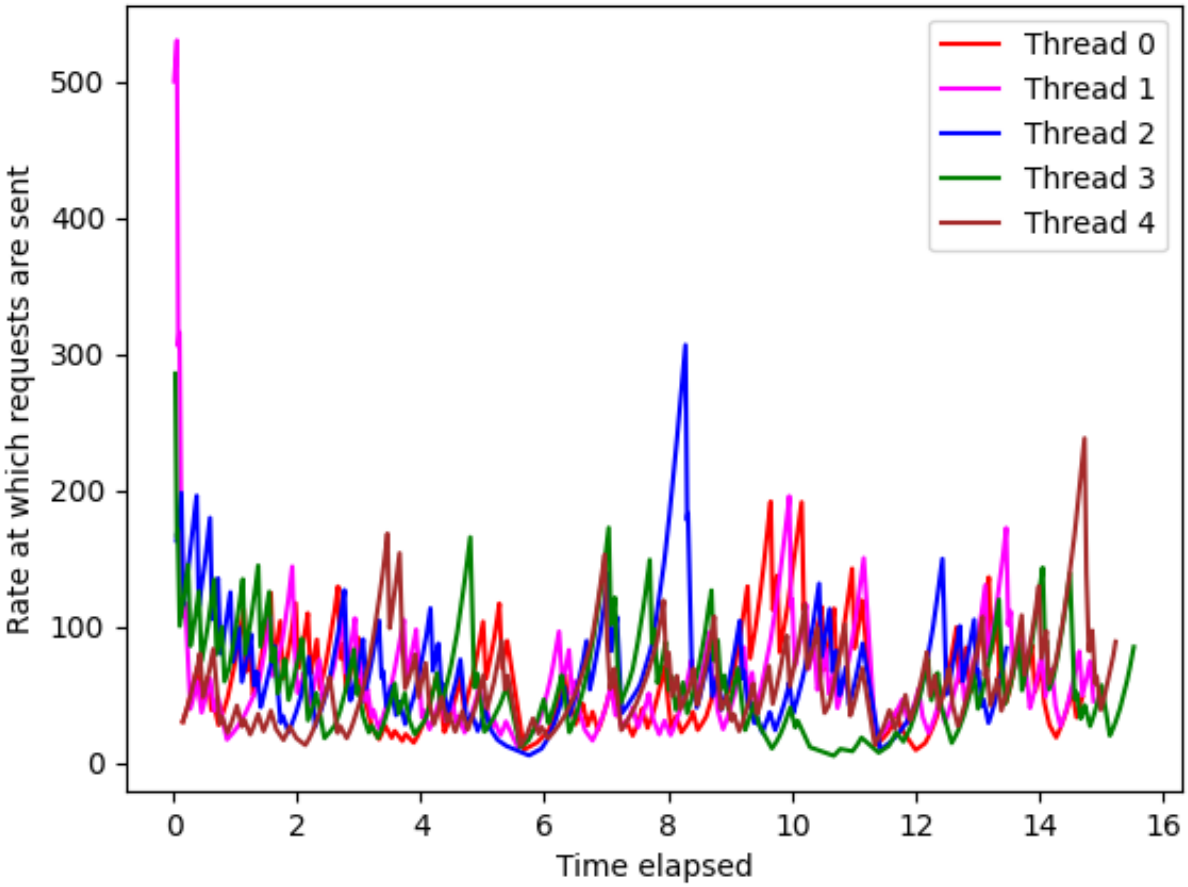


Figure 4: Variation of rate of sending requests with time

As we can see, first the rate drops to a stable value, then oscillates near it. However sometimes it is the case that the threads compete against each other, which is analogous to the traffic on a server due to multiple clients, and in such cases, even though AIMD makes sure to restrict one thread (client in our analogy) to take over, sometimes it is possible due to randomness that one client (for example blue here) momentarily starts getting a larger share of the bandwidth, and its repercussions are the reduced rates of other threads (clients) as we can see in this case, when blue peaks, other threads show a decline in their rates. But in the long run, we see that each thread on averag receives equal share of the bandwidth, which is a good thing!

5 Week 3

In this week, we planned to figure out the optimum value that we should give to socket.timeout, this can be done by estimating the Return Trip Time of our requests. We found this out using the **EWMA** strategy.

After each successful requests, we calculate the RTT for this request, and then we update the Estimated RTT using the following equation -

$$Estimated\_RTT = 0.875 * Estimated\_RTT + 0.125 * Current\_RTT$$

We also maintained a variable which denotes the variation in Estimate.RTT and Current_RTT, and that is updated using the following equation -

$$Deviation\_RTT = 0.75 * Deviation\_RTT + 0.25 * |Estimated\_RTT - Current\_RTT|$$

Now, we updated timeout at each successful request as follows -

$$Timeout = Estimated\_RTT + 4 * Deviation\_RTT$$

In case of unsuccessful requests, we multiplied timeout by a factor of 2 so that our next requests has more chances of success. Initial timeout and Estimated RTT was kept to be 0.01.

Apart from this, we followed the same strategy of AIMD as we did in Milestone 2, as it adapts well to the variable rate servers as well.

We were taking on average $\sim 30s$ on the servers hosted and under 100 penalty.

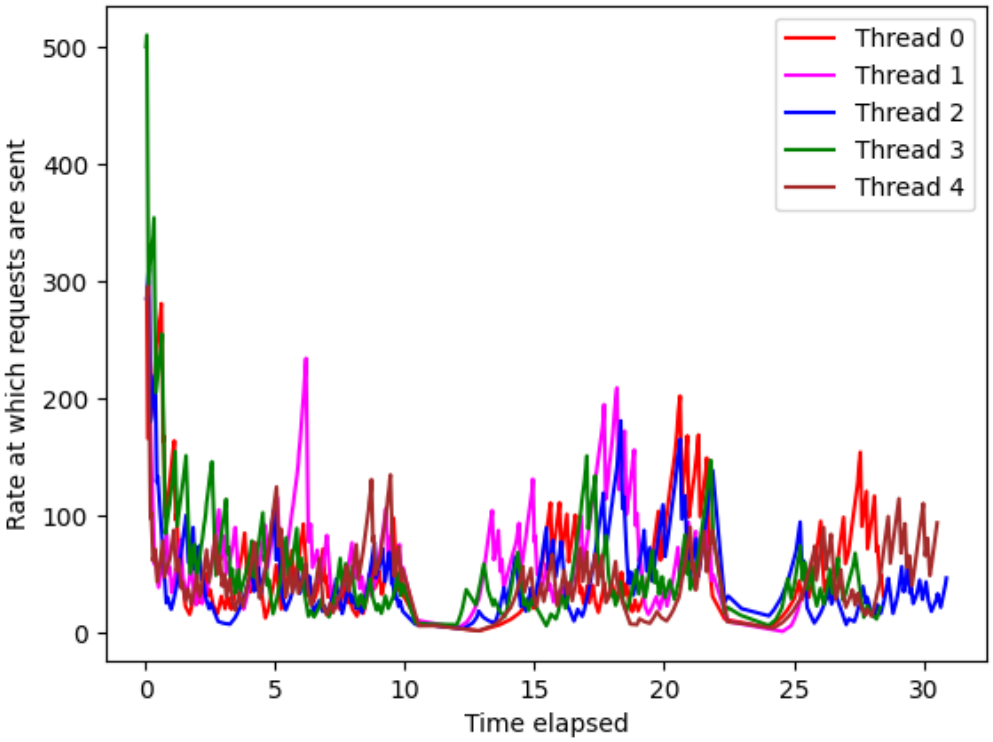


Figure 5: Rate of sending requests vs time

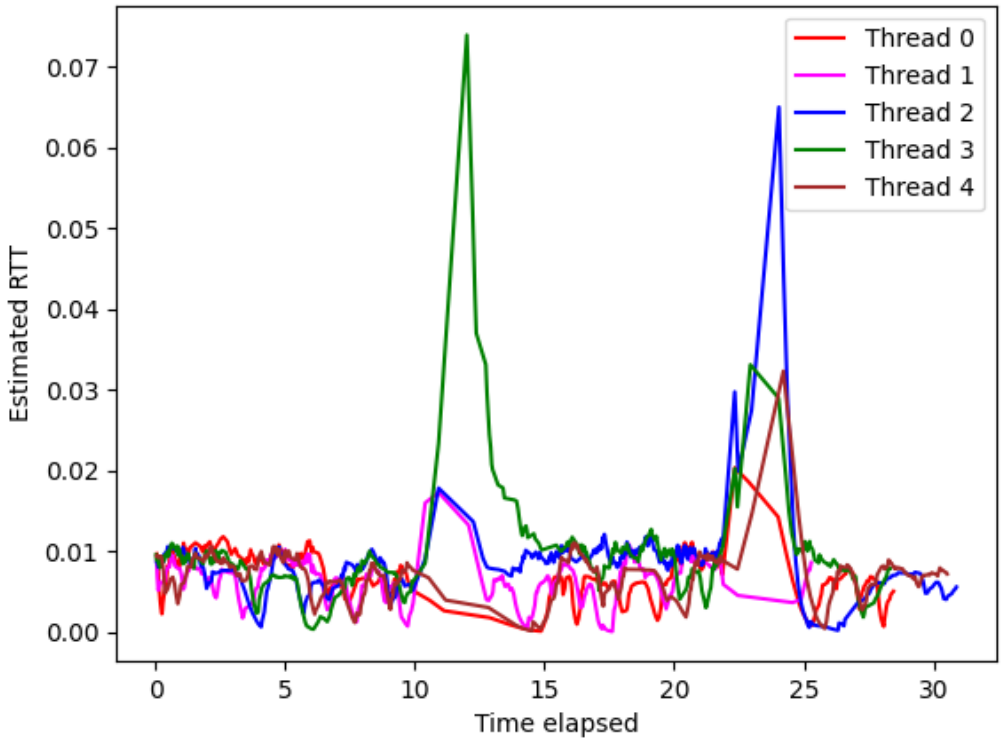


Figure 6: Estimated RTT vs time

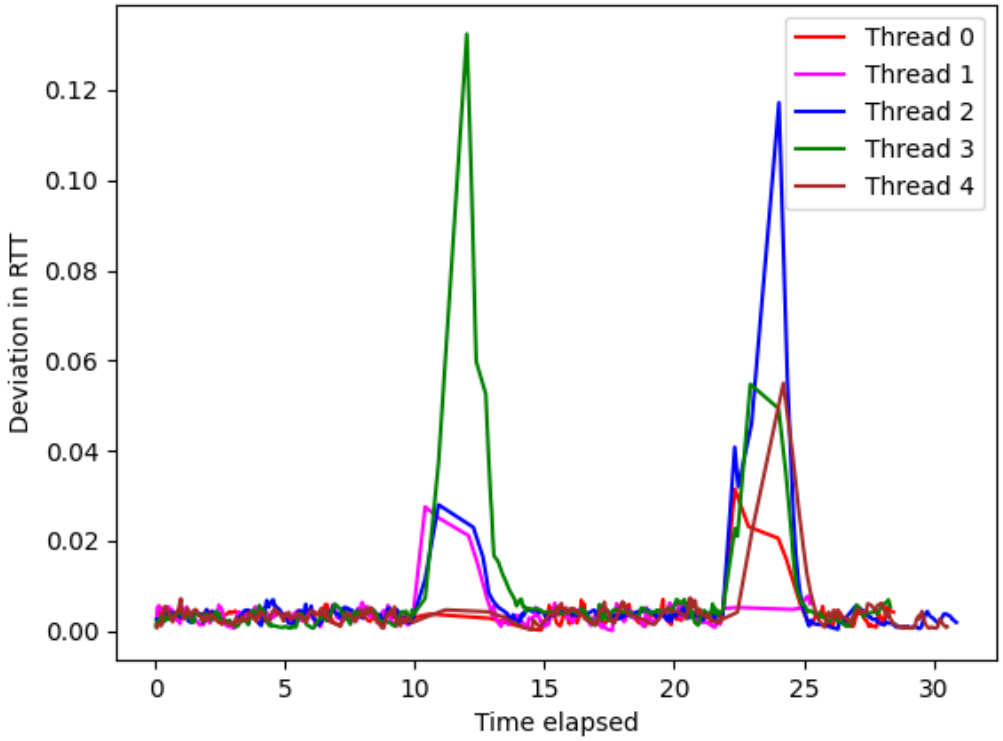


Figure 7: Deviation in RTT vs time

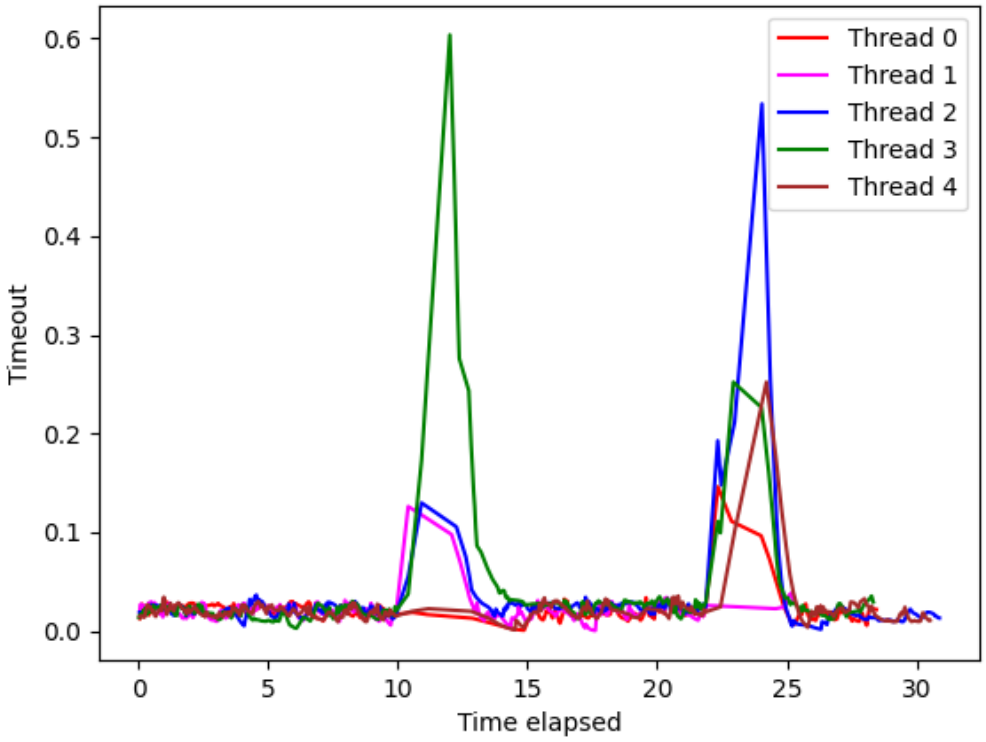


Figure 8: Variation in socket timeout vs time