COL334 Assignment 3
(Milestone 2)

Tript Sudhakar (2021CS10110)
Parth Patel (2021CS10550)

November 1, 2023

Contents

§1 Logic of the client program

§1.1 High level overview and design

- We have designed the program using multi-threading (two threads).
- First thread (sending thread) handles sending requests. We are sending a burst of requests of size `burst_size` per `sleep_time`. Second thread (receiving thread) handles receiving data.
- We first wait for the second thread to complete (i.e. receive all the data) and only after this, is the first thread finished.
- At the end, the client sends the MD5 hash value for the complete data received and we get the appropriate feedback from the server.
- Note that rtt in this document refers to the return trip time (client to server plus server to client) and not to be confused with the round trip time for TCP.

§1.2 Receiving data reliably as well as fast

- Whenever the sending thread sends too much requests (even though the leaky bucket had no tokens left), the receiving thread will keep track of the replies received and a at a certain threshold percentage of losses from the server, it decides to decrease `burst_size` by a factor of 2.
- Whenever the sending thread sends requests at a lesser rate, the receiving thread, which is receiving all the data which was sent to the server, will increment the `burst_size` by 1. This will in turn increase the number of requests that the sending thread is requesting to the server in one *burst*.
- We have used lock when a shared resource is modified by two processes and both the processes are actively using the value of that object as well.
- Since in this checkpoint, the server is of variable rate, we also update the rtt. We have used the EWMA (Exponentially weighted moving average).

$$rtt_{new} = \alpha * sample_{new} + (1 - \alpha) * rtt_{old}$$

- The `sleep_time` will be kept as `burst_size*rtt`. Also, we have kept an additional threshold so that `sleep_time` doesnt get below a minimum value.

§2 Code structure and implementation

§2.1 Basic constructs

- Commands like `send_size`, `submit`, `message` and variables/objects like `size`, `burst_size`, `sleep_time`, `lines_recv`, `index`, `datastring`, `hashval`, `bytestr`, `feedback`, `lock`, `rtt`
- Data structures : `offset_list` list, `received` list, `data_list` list, `sent_time_list` list, `recv_time_dict` dictionary, `burst_size_list` list, `burst_time_list` list
- The client always requests for 1448 bytes from the predetermined disjoint offsets which are calculated from the response of the `SendSize` request (except possibly the last offset).

§2.2 Receiving and Sending Threads

- The threads `sending_thread` and `receiving_thread` handle concurrent communication with the server, associated with the functions `sending_process` and `receiving_process`
- `receiving_process` receives the messages from the server, processes it and stores it in `data_list` list.
- `sending_process` sends requests to the server and waits for `sleep_time` amount of time after each request. In case of congestion (leaky bucket), the `sleep_time` will be incremented by `receiving_process`, hence ensuring robustness.
- `to_recv` denotes the number of lines which are *heard* by the receiving process. The protocol we have used is that if `to_recv/burst_size` is greater than 0.8 (a heuristic) then we increment the `burst_size` by 1 else we decrease it by a factor of 2. This protocol is very similar to the TCP's AIMD (Additive increase multiplicative decrease) protocol.
- `lock` is used in the `receiving_process` which makes sure that when the `burst_size` changes, the `sending_process` doesn't proceed with the old `burst_size`.

§2.3 Main program

- The order of execution of threads is
  1. `sending_thread.start()`
  2. `receiving_thread.start()`
  3. `receiving_thread.join()`
  4. `sending_thread.join()`
- The main program finally constructs the final string and computes its MD5 hash value to and submits that to the server.

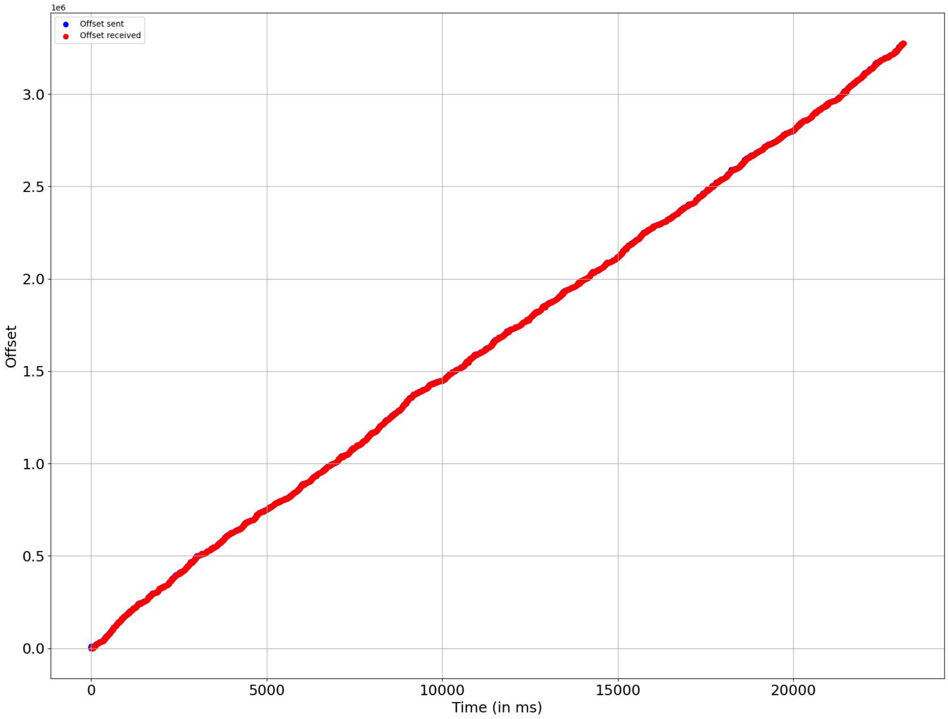
§3 Results and Analysis

We performed the analysis for 50000 lines for the local server and the vayu server (10.17.7.218) as they both give similar no of packets to be received (around 2200).
In each of the two servers that we analysed our client program on, we plotted five graphs: `burst_size` vs time, sequence-number trace and finally, a zoomed-in version of the sequence trace, `sample and rtt_mean` vs `lines_recv` and a zoomed-in version of `sample and rtt_mean` vs `lines_recv`.

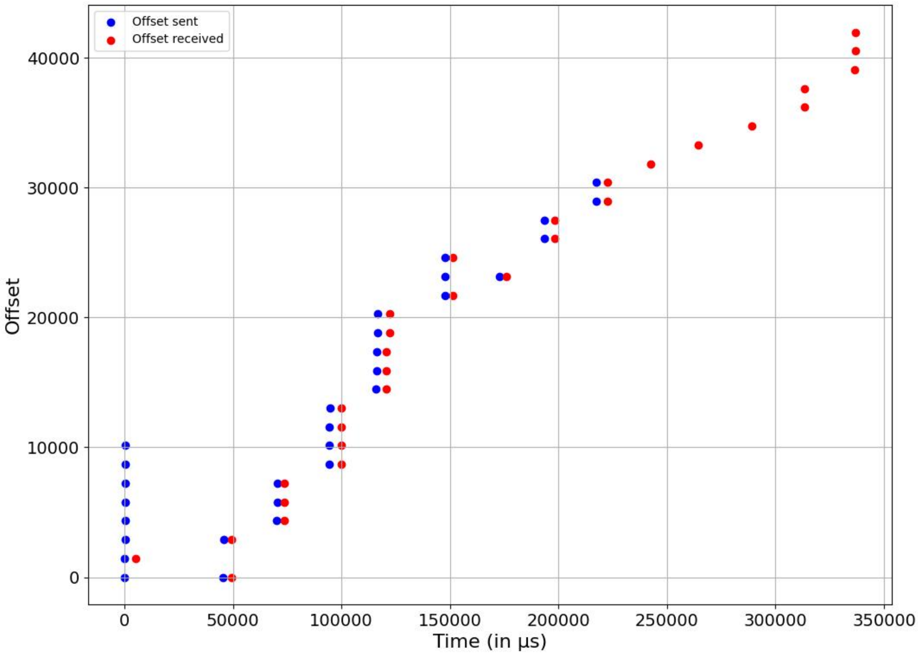
§3.1 Plots

§3.1.1 Vayu Server

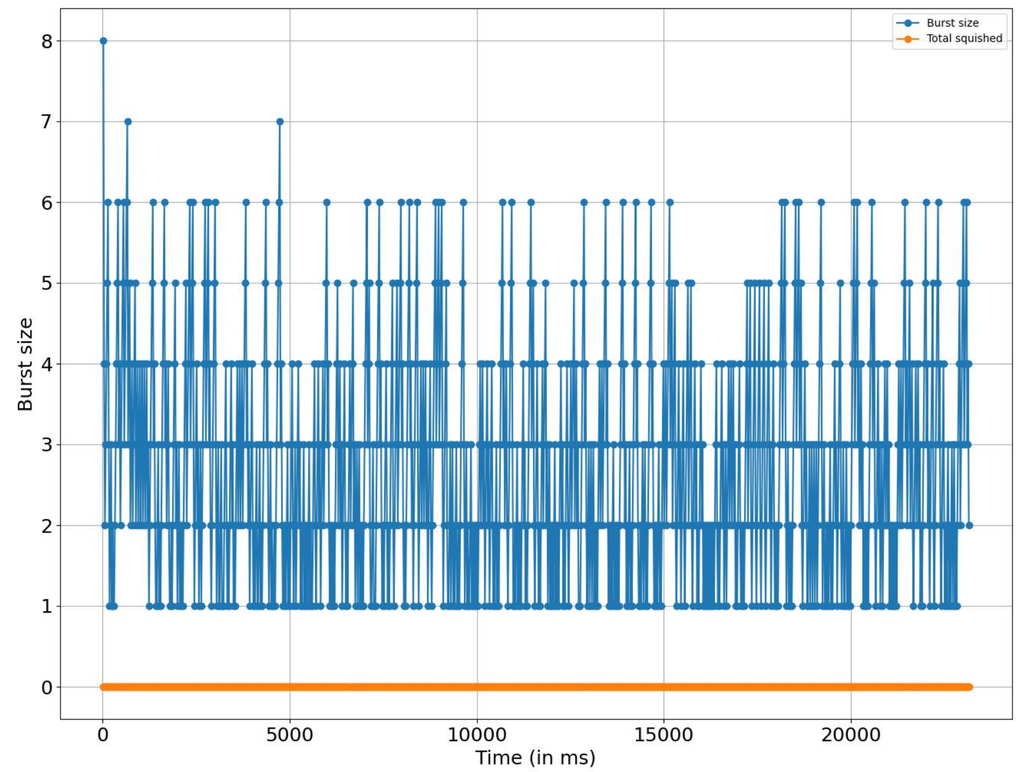
Sequence number trace



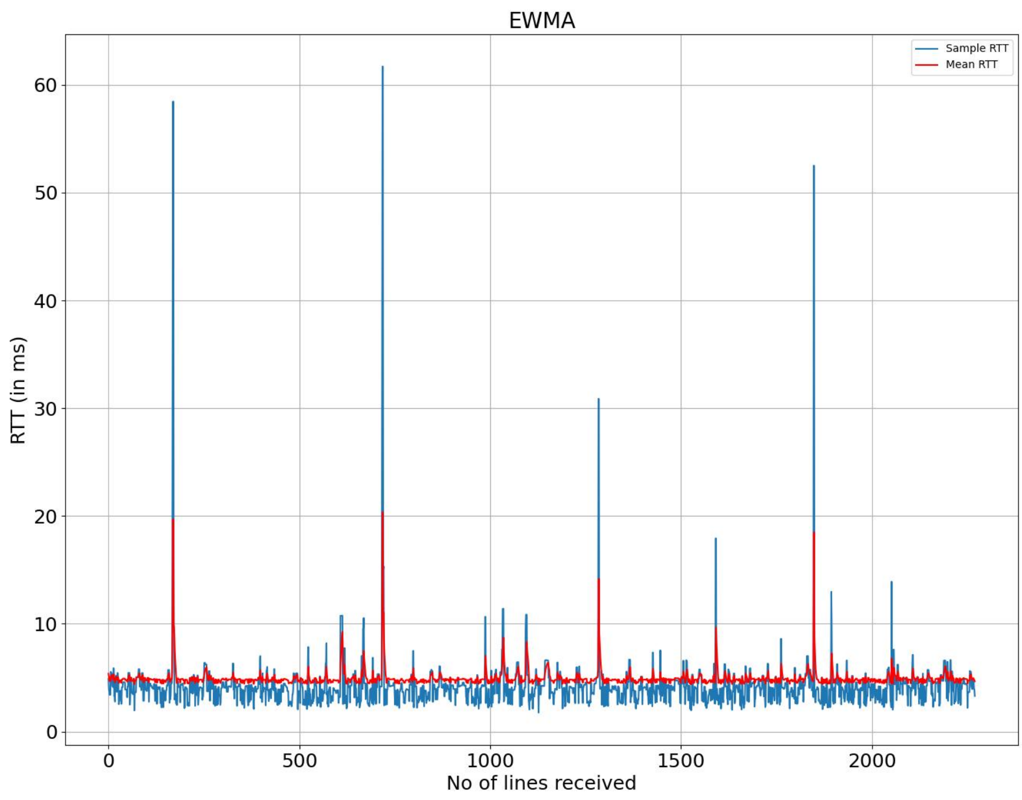
Zoom-in sequence number trace



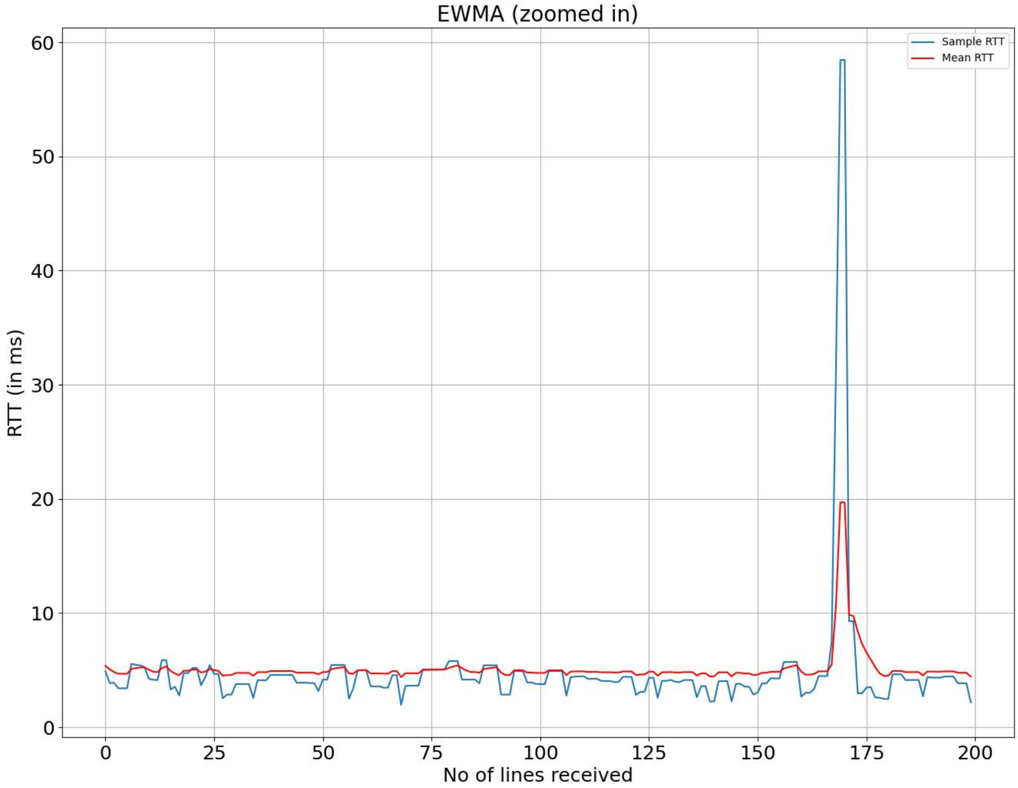
Burst size vs time



RTT vs lines received

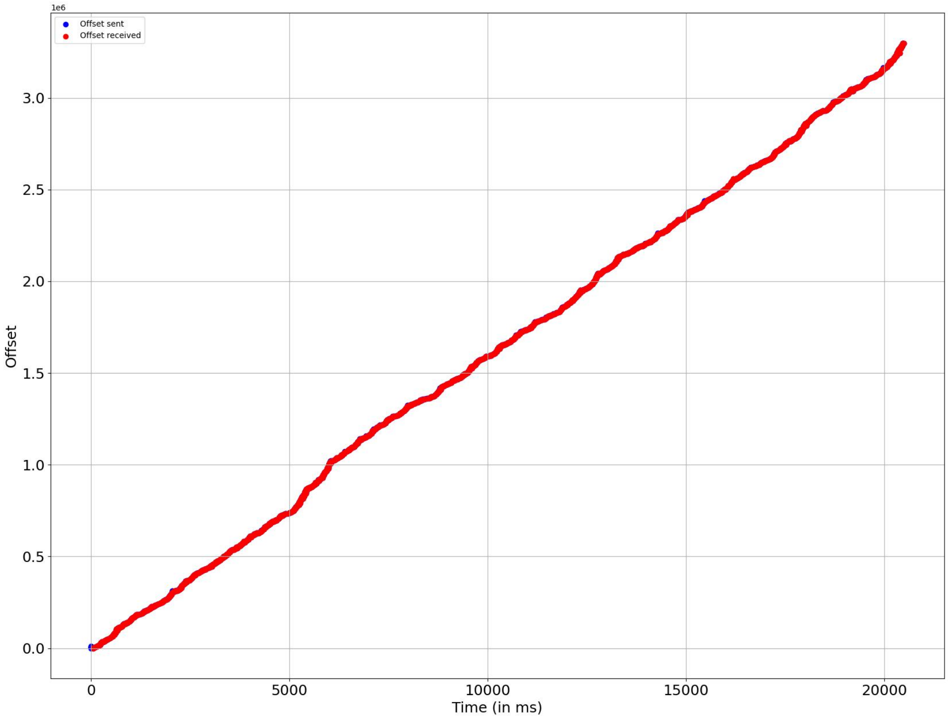


RTT vs lines received (zoomed-in)

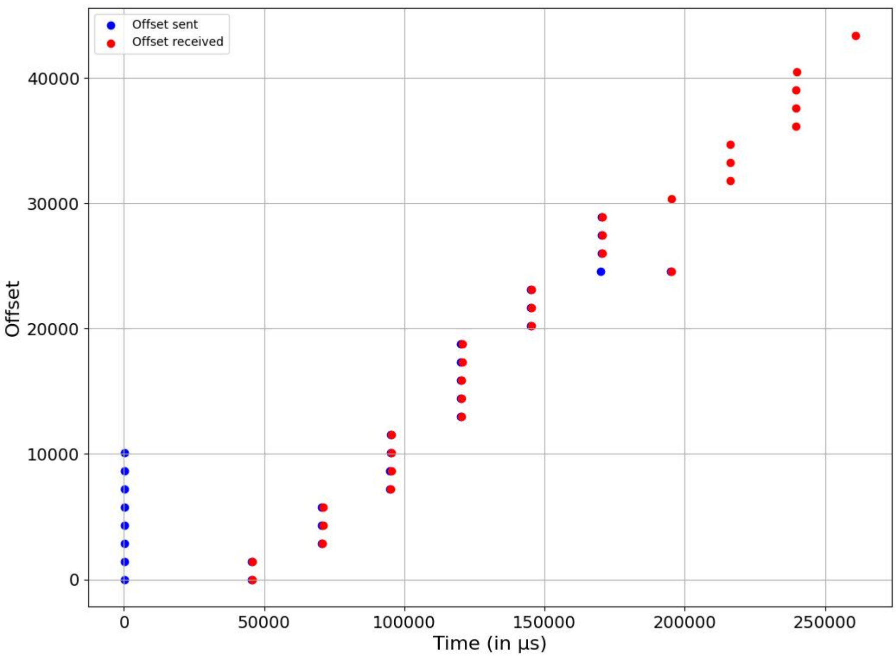


§3.1.2 Local Server

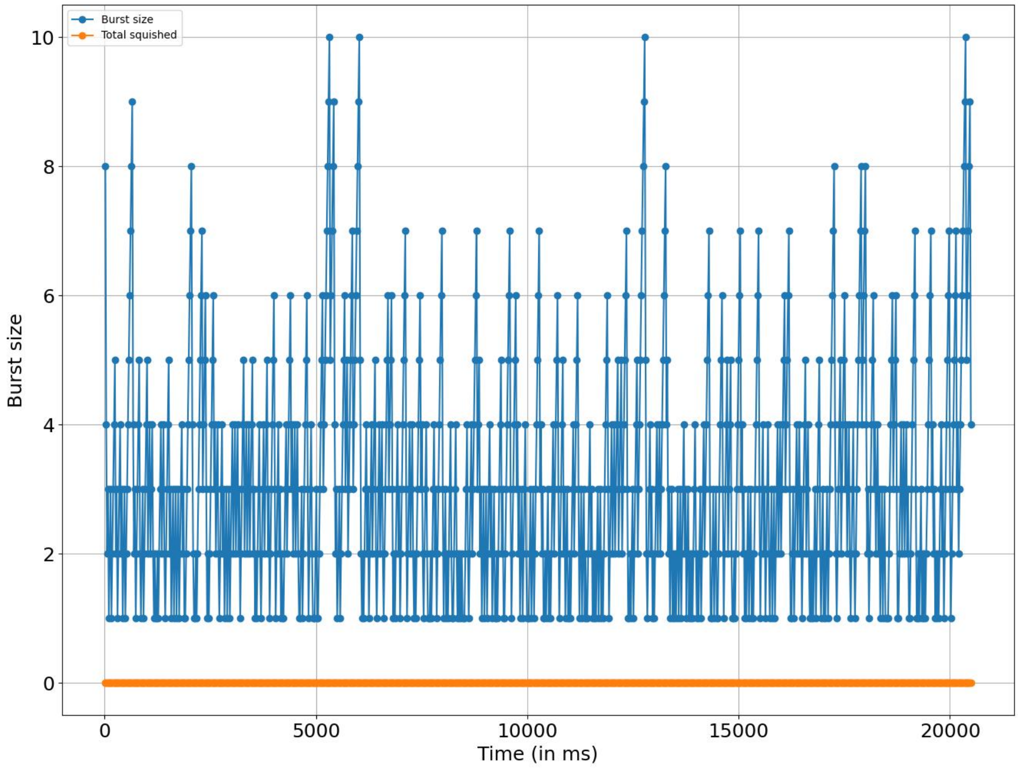
Sequence number trace



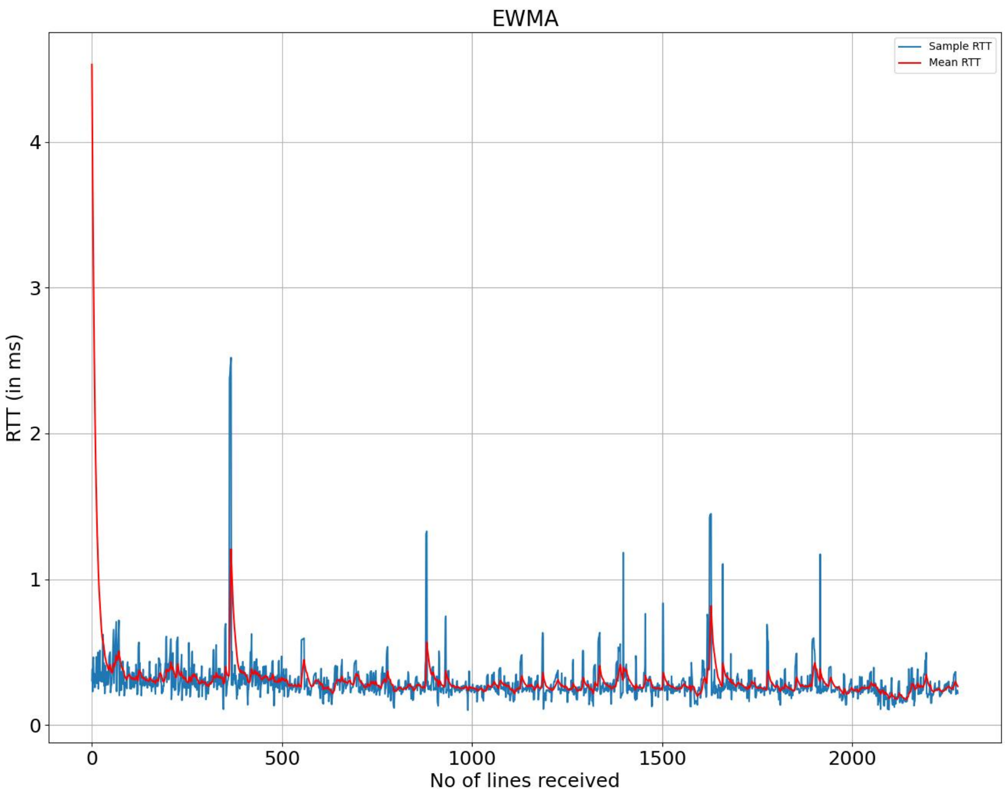
Zoom-in sequence number trace



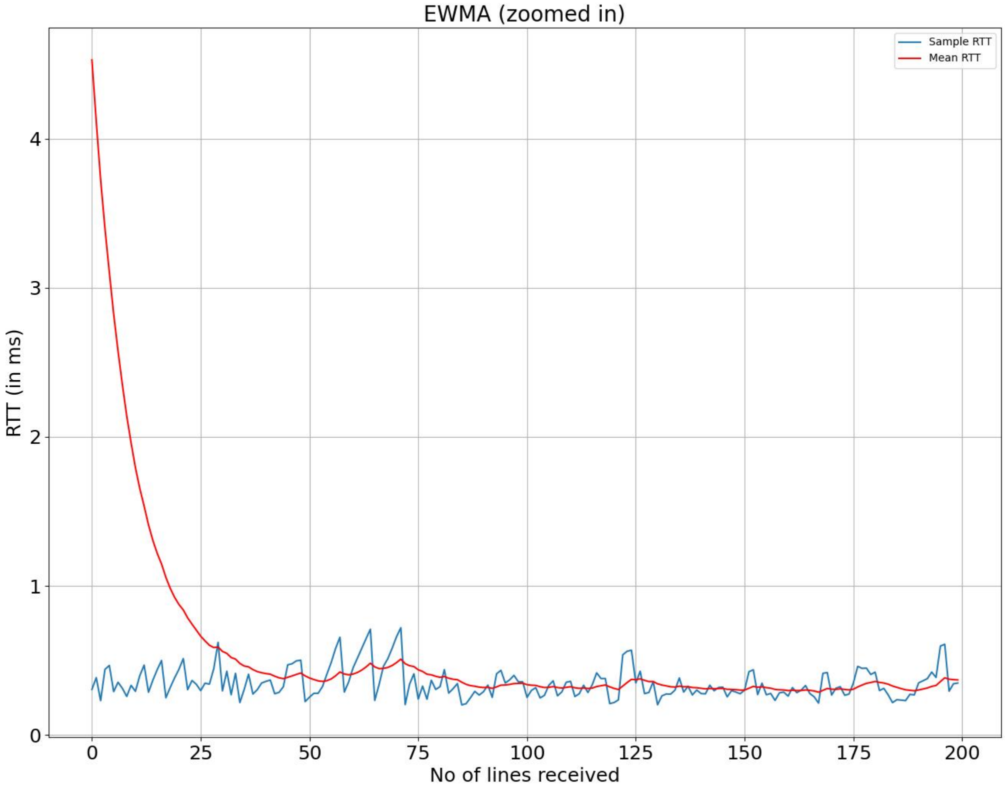
Burst size vs time



RTT vs lines received



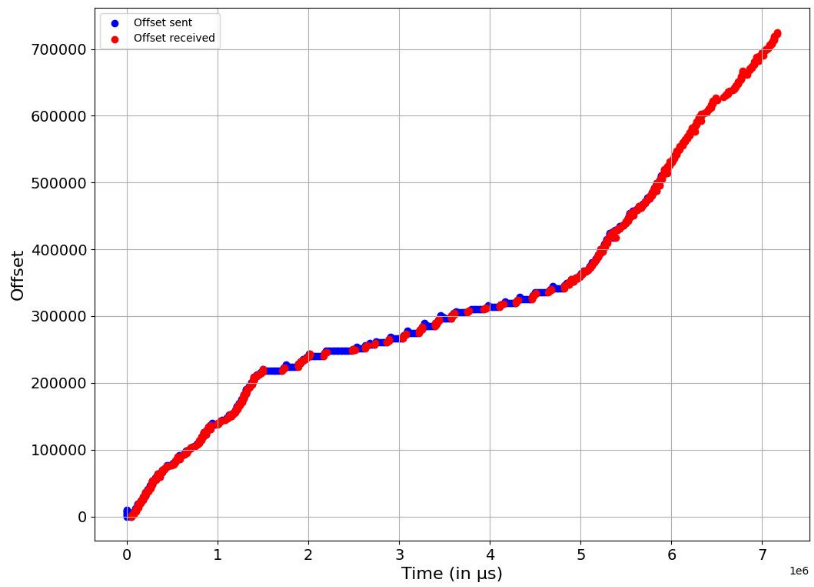
RTT vs lines received (zoomed-in)



§3.2 Observations and Conclusions

1. We tried experimenting with various values of α and the cutoff minimum `sleep_time`. The value of α that we finally found *'optimal'* was around 0.2, and the cutoff minimum `sleep_time` as 0.02 seconds. The above plots are also made for $\alpha = 0.2$.

2. Here is a *not-so-zoomed-in* version of the sequence number trace (obtained by localhost server) -



We can observe that the rate is actually varying periodically. Slow rate from around 1.5 sec to 4.8 sec and then a fast rate before 1.5 sec and after 4.8 secs.

3. We always get the penalty as less than 35, mostly it is around 10-30. The average rtt for vayu (other servers) is around 0.007s (more analysis on this was done in the previous checkpoint report). We have also successfully implemented the client which never squishes.

4. We have plotted sample rtt and mean rtt as a function of number of lines received. We can see that the mean rtt plot is a smoothed version of the sample rtt. We also observed that as we decrease the value of α, the mean rtt plot smooth ens more and more.

5. While sending the requests to the server in bursts, we always send the packets from the beginning which have not been received yet as shown by the sequence trace.

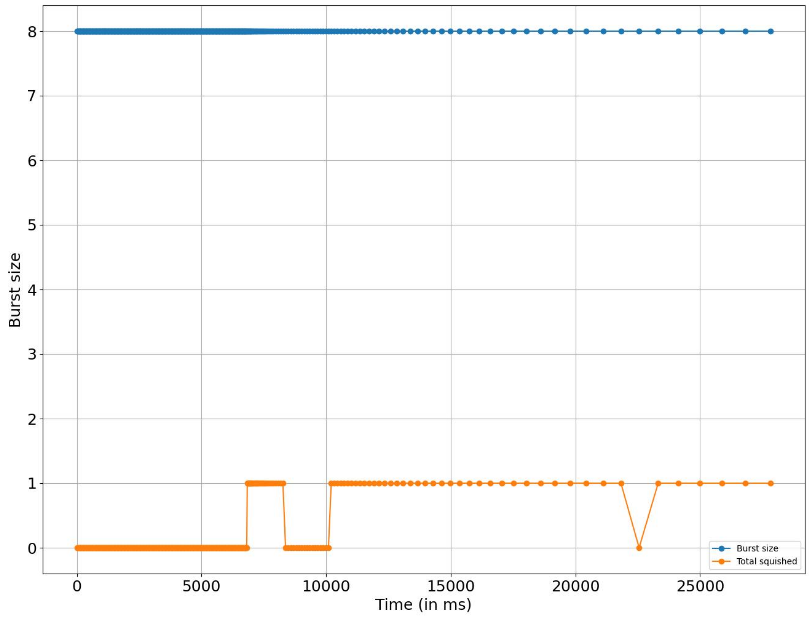
6. In the zoomed-in sequence trace, we can see that the blue dots represent the requests sent in bursts and the red dots near these blue dots corresponding to that particular burst represent the replies received from the server.

7. We can see that whenever there are as many red dots as there are blue dots in the burst (the red dots all overlap with the blue dots in the burst), the burst size increases in the next iteration. Similarly, if some request doesn't receive a reply i.e there is no red dot corresponding to that blue dot, the burst size halves in the next iteration.

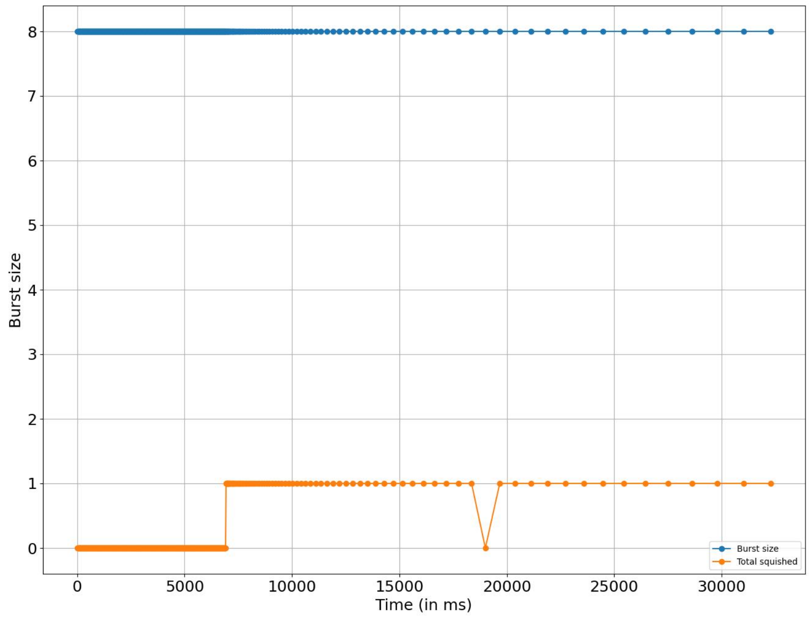
8. The burst size vs time graphs also depict the classic saw-tooth behaviour of the AIMD client. The no of squished is also zero throughout the process for both the servers.

9. We have also tried using constant burst size of 8 but as we can see from the plots, it results in squishing. Note the sparseness as we are getting squished.

Burst-size trace for a constant burst client with vayu server

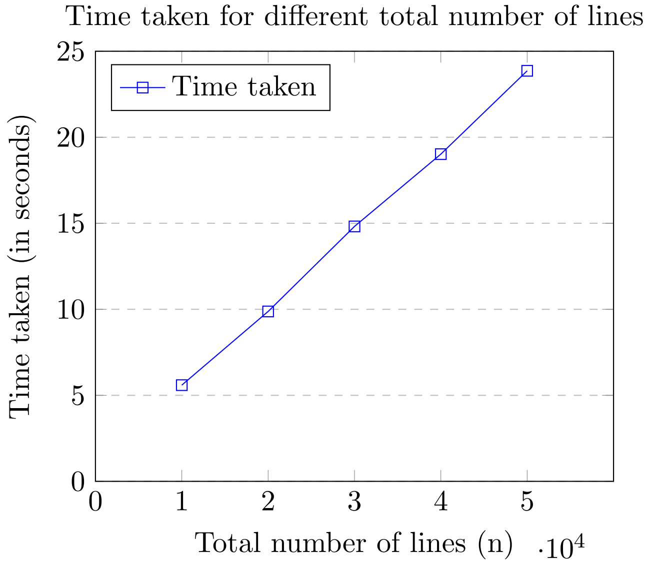


Burst-size trace for a constant burst client with local server



§3.3 Time taken by varying total number of lines

Total number of lines	Time taken (in s)
10000	5.592
20000	9.875
30000	14.817
40000	19.019
50000	23.863



- By running the localhost server, we observe that the time increase linearly with the number of lines, which is what we expect.

§4 Contribution

Both members contributed equally in terms of code/implementation and report making.
Parth Patel (2021CS10550) - 50%
Tript Sudhakar (2021CS10110) - 50%