COL334 Assignment 3 (Milestone 2)

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

October 25, 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.

§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.

§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`
- 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. If the message contains "squished", it increases the `sleep_time` by `0.001s`
- `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 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.

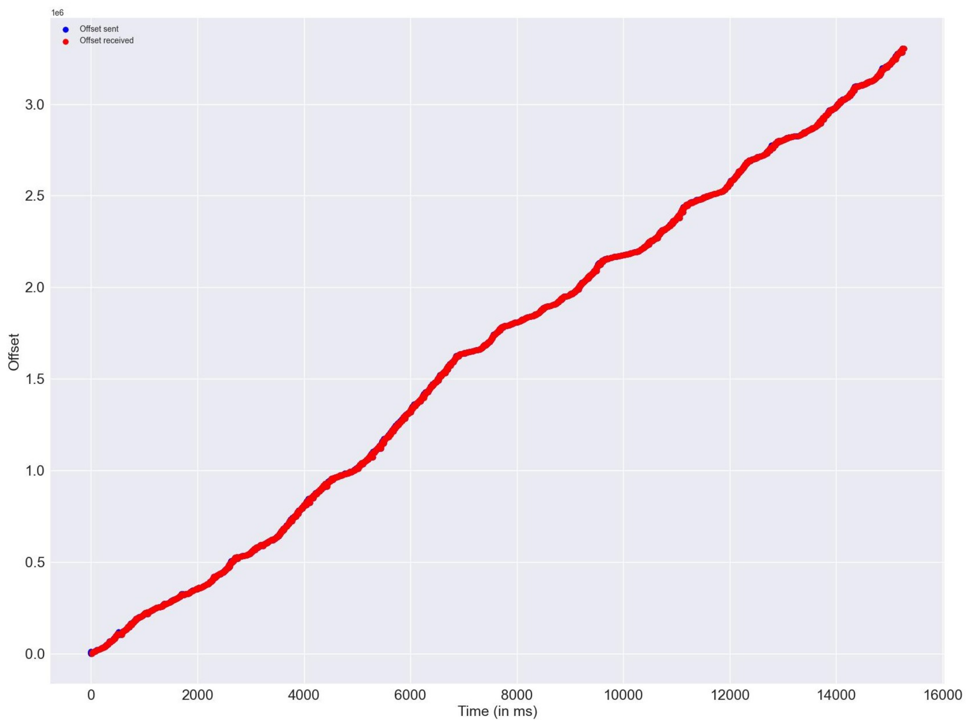
§3 Results and Analysis

We performed the analysis for 50000 lines for the local server and the vayu server (10.17.7.34) 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 three graphs: `burst_size` vs time, sequence-number trace and finally, a zoomed-in version of the sequence trace.

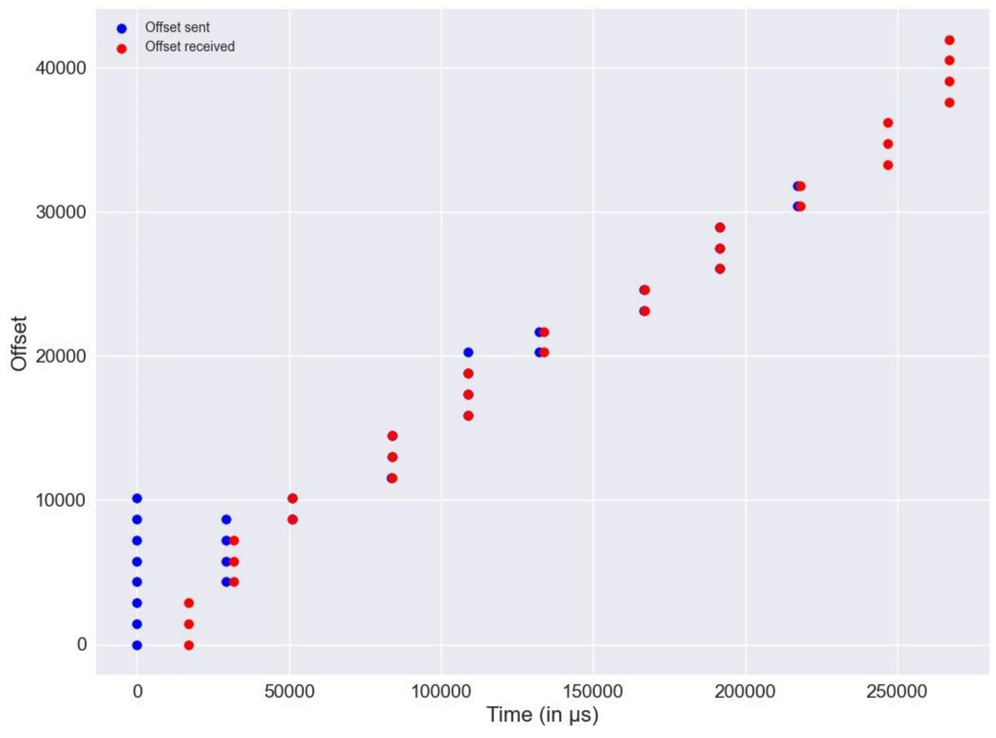
§3.1 Plots

1. **Vayu server**

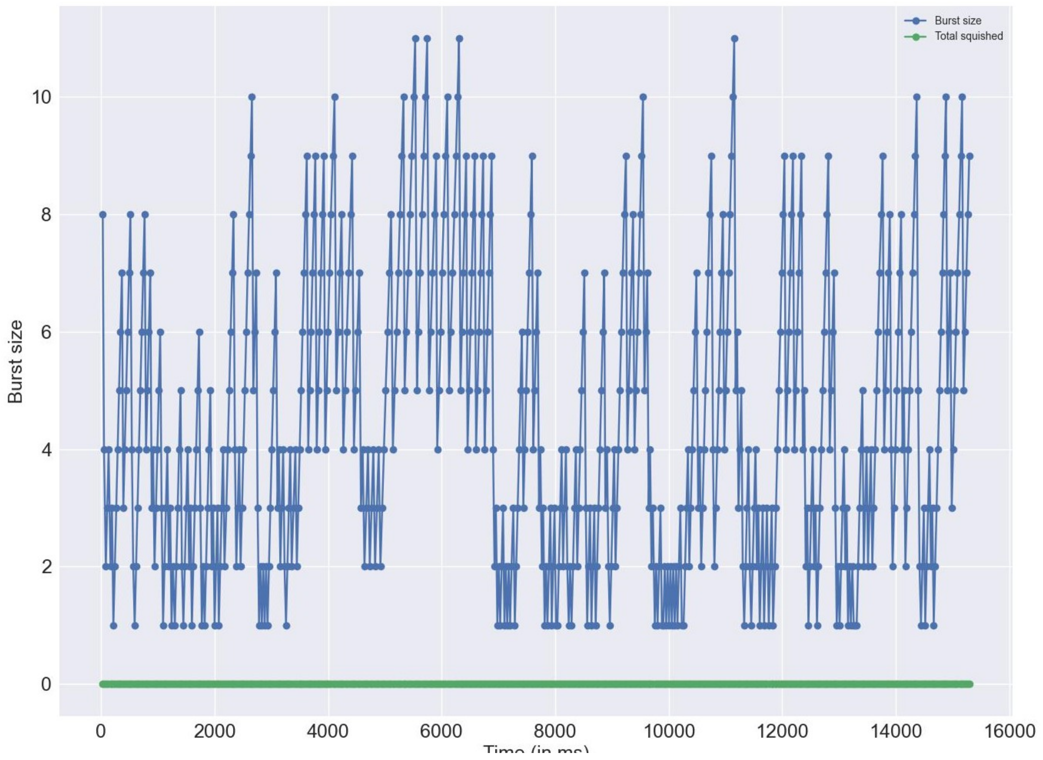
   a) Sequence number trace



   b) Zoom-in sequence number trace

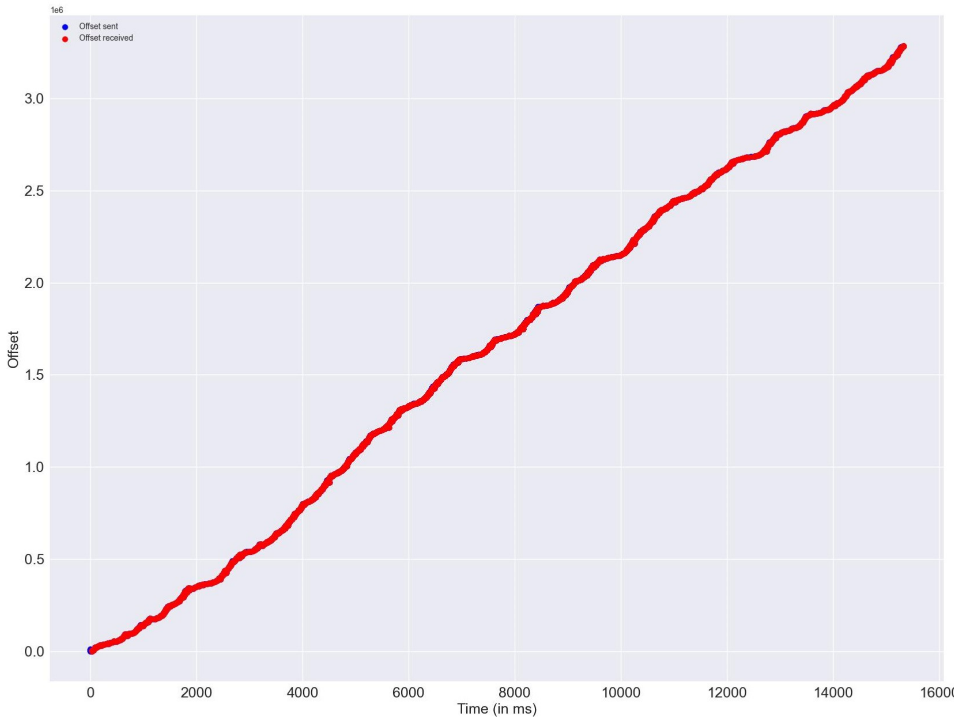


c) Burst size vs time

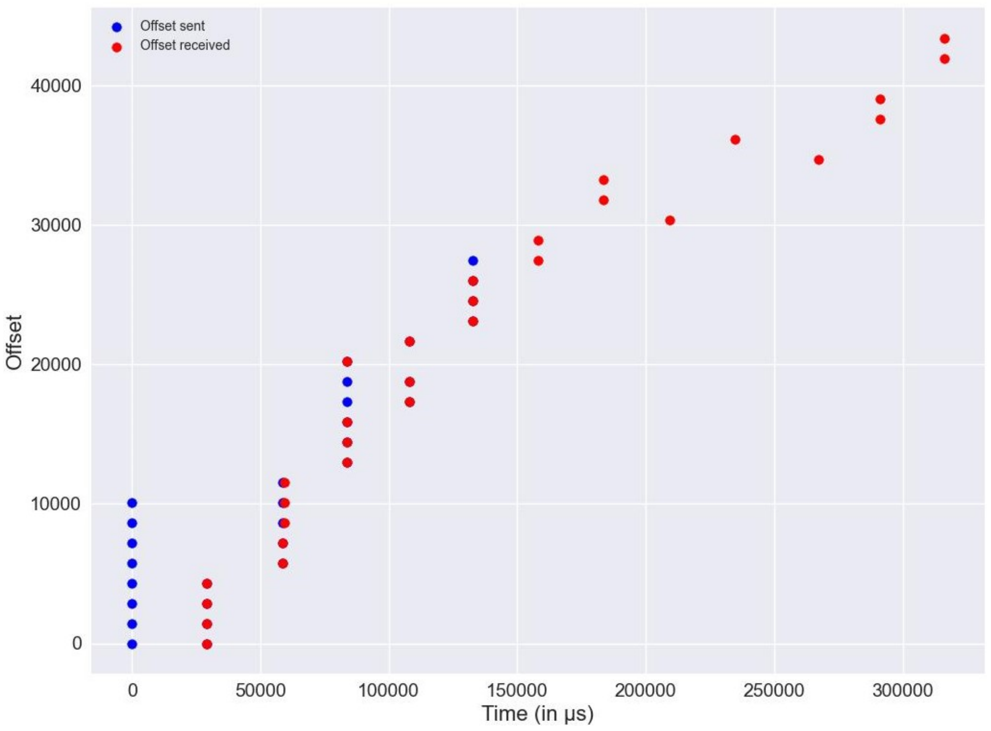


2. **Local server**

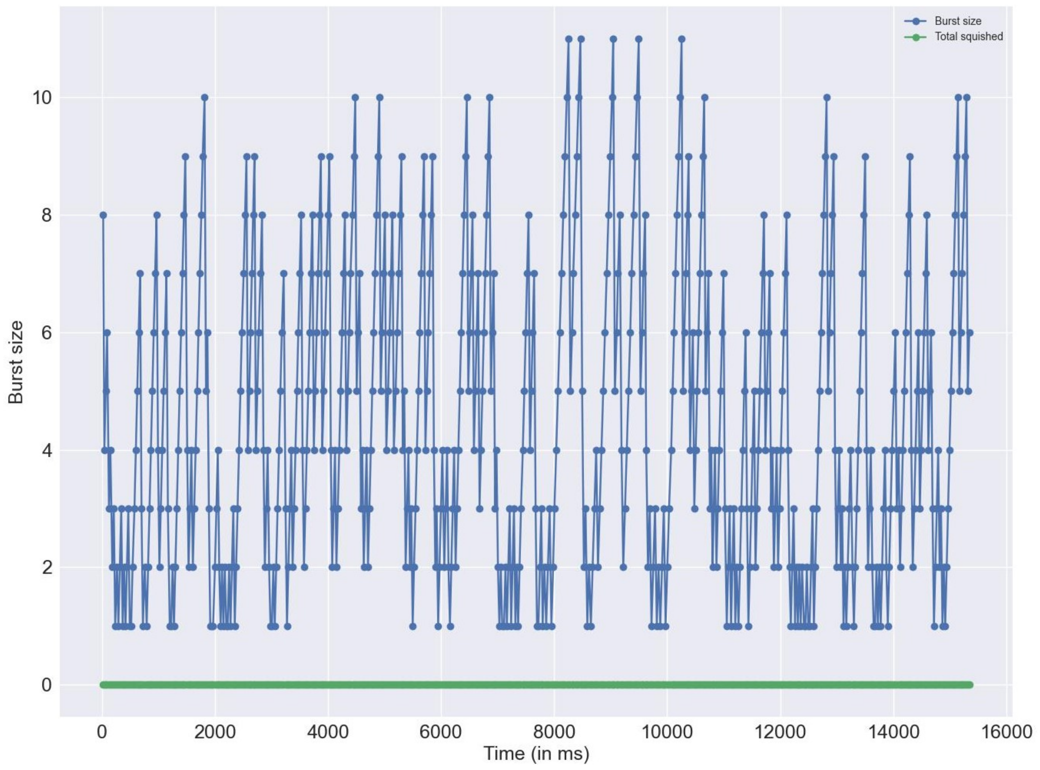
a) Sequence number trace



b) Zoom-in sequence number trace



c) Burst size vs time

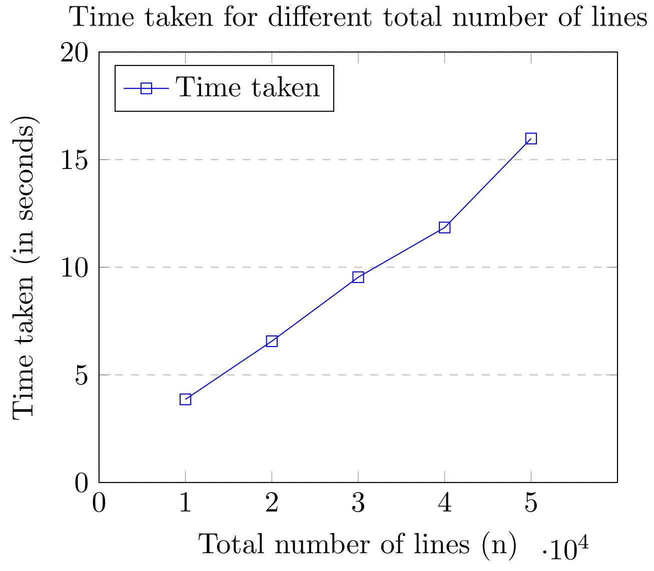


§3.2 Observations

1. 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.
2. 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.
3. 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.
4. For example, in the first burst, none of the replies are received. So, the burst size is halved to 4. In the next 2 bursts, all the replies are received. So, the burst size increases to 6. In the next iteration, only 4 replies are received, which causes the burst size to be halved to 3 ans so on.
5. 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.

§3.3 Time taken by varying total number of lines

Total number of lines	Time taken (in s)
10000	3.862
20000	6.564
30000	9.539
40000	11.846
50000	15.980



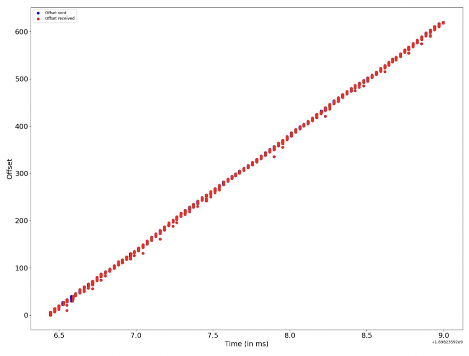
- We observe that the time increase linearly with the number of lines, which is what we expect.

§4 Hecker Approach :)

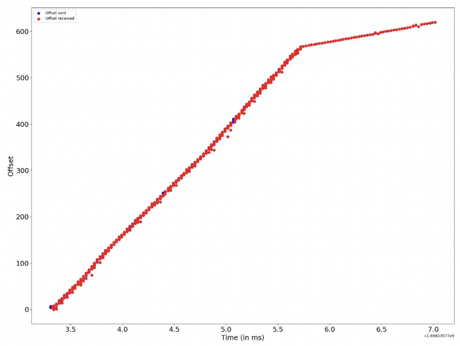
Hack : Since we can know the leaky bucket rate for the iitd vayu server by experimentation, we can choose to make a constant `burst_size` and constant `sleep_time` client. Note that here our receiving thread is received the replies from the server continuously. Some of these results (which are locally optimal) are summarised in the table below (by varying `burst_size` and `sleep_time` for 10000 lines on local server)-

Figure	`burst_size`	`sleep_time` (in s)	Time taken (in s)	penalty
a	8 (not squished)	0.023	2.571	63
b	8 (squished)	0.022	3.739	418
c	12 (not squished)	0.036	2.704	89
d	12 (squished)	0.035	3.359	298

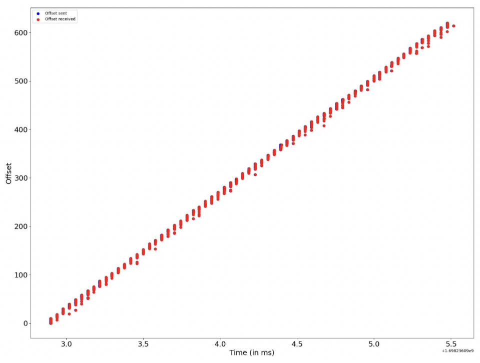
(Typo - the figures below should have x axis scale in seconds and not ms)



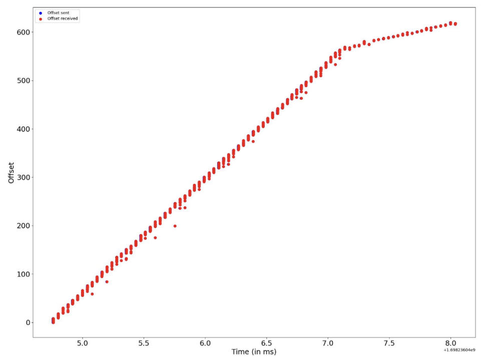
(a)



(b)



(c)



(d)

- The optimum parameters that we got for `burst_size` as 8 was `sleep_time` as 0.023s to *just* not get squished. And similarly for `burst_size` as 12, `sleep_time` should be kept as 0.036s to *just* not get squished.
- From this experiment we can conclude that the constant leaky bucket rate is around 8/0.023 or 12/0.036 which are both close to around 340 lines per sec, and given that we used 1448 bytes, it is around 340*1448 = 500KBPS. Such an analysis can be performed for any such *constant leaky bucket network* and hence construct and optimal client which is very rapid,reliable and robust.

§5 Contribution

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