Computer Networks

Assignment3

by Mani Sarthak & Harshit Gupta

October 15, 2023

Some points to be considered:

- All the codes are managed on **GitHub** and can also be found in this **folder**.
- For milestone 1 we had applied the following strategies to get the data.
  - Naive approach to send request sequentially with incrementing offset until get the data for that offset.
  - Sending and receiving data in bursts with a fixed waiting time window for both part.
  - Using **multithreading** to parallely send request and receive data.
  - **AIMD** implemetation for variable rate server was also implemented.

  However all analysis have been done on the approach using multithreading only.
- Large data graphs were plotted using **matplotlib** library.

Token Distribution:
1. Mani Sarthak : 2021CS10095 : 10
2. Harshit Gupta : 2021CS10552 : 10

§1 Logic Overview

The code is designed to download and verify data from the server using **multi-threading**. It handles receiving data, tracking offsets, and sending requests concurrently to optimize data retrieval. The MD5 hash calculation and submission ensure data integrity.
In addition, the code maintains an array that is pre-computed before sending requests for fetching data. This array contains offset-packetpairs and receivedpairs for bookkeeping. This bookkeeping mechanism is used to keep track of which data has been received and to manage the offset and packet size for efficient data retrieval.

§2 Code overview

§2.1 Implementation

1. **Initialization**:
   - The server's address and various constants are set.
   - Helper functions for file writing, data extraction, and size parsing are defined.
2. **Receiving and Sending Threads**:
   - Two threads are created to handle concurrent communication with the server: `recieving_thread` and `sending_thread`.
   - `recieving_thread` listens for data from the server, extracts the payload, and stores it in a dictionary indexed by the data's offset. It also keeps track of received data using a list.
   - `sending_thread` sends requests to the server based on a list of offsets and sizes, and it ensures proper synchronization when requesting data.
3. **Main Function**:
   - The `main` function starts the application.
   - It first queries the server to determine the size of the data to be received.
   - It initializes a list of offsets and sizes based on the maximum packet size.
   - Two threads are started: `sender_thread` and `reciever_thread`, which operate concurrently.
   - After both threads complete, the `main` function reconstructs the received data and calculates an MD5 hash of it.
   - It then sends the MD5 hash back to the server for verification.
4. **Execution**:
   - The program executes the `main` function if it is run as the main script.

§2.2 Code flow

In code implementation , we created a main function in which first we send the **SendSize** request to get the number of bytes received and then we set the numbytes to send each time , we calculates number of requests and corresponding offsets with this information. Then we start the sending and recieving threads and wait till they are complete and store data in form of dictionary. Once they are complete we compile the data in form of stirng and then we convert it into **MD5 hash** using **hashlib** library and the submit the data to get the result.

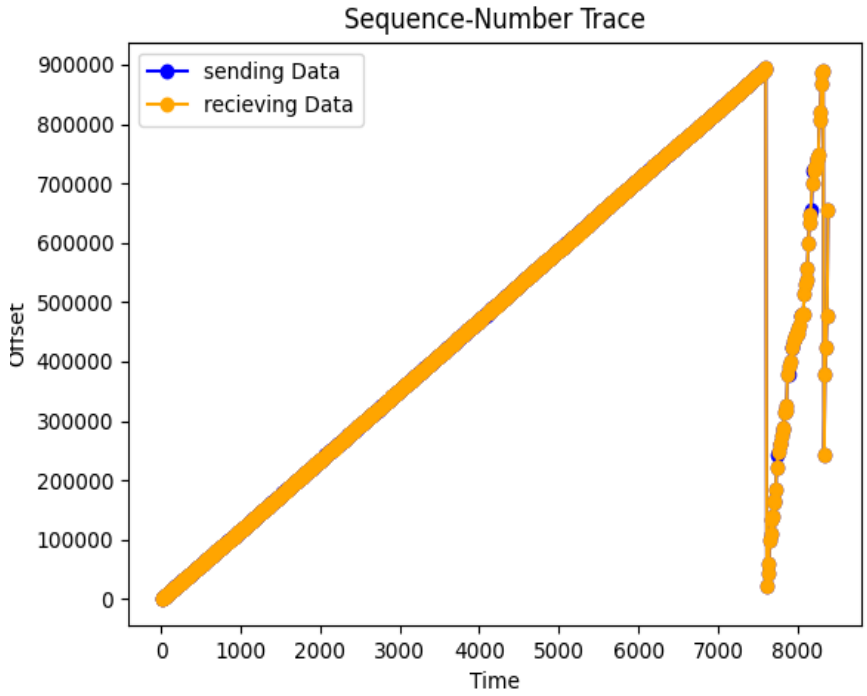
Below is the code segment, we only included the main function of the code to give the high level overview.

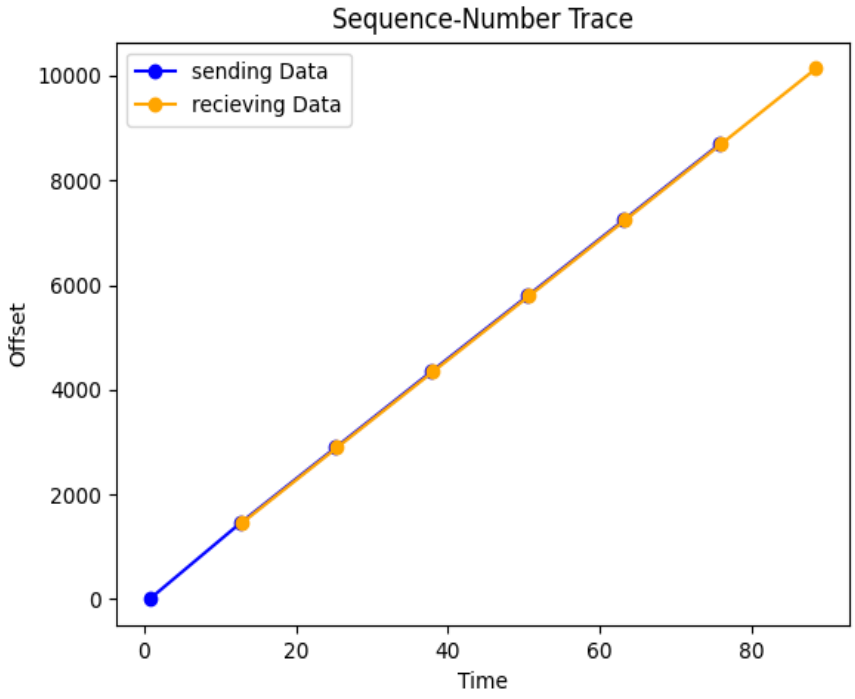
```
83  def main():
84      try:
85          message = 'SendSize\nReset\n\n'
86          sock.sendto(message.encode(), server_address)
87          data, server = sock.recvfrom(2096)
88          data = data.decode()
89          byte_size = getSize(data)
90
91      finally:
92          print('Done')
93      arr = [[x, min(x+maxSize, byte_size)-x] for x in range(0, byte_size, maxSize)]
94      requests = len(arr)
95      sender_thread = threading.Thread(target=sending_thread, args=(requests,arr))
96      reciever_thread = threading.Thread(target=recieving_thread,args=(requests,arr))
97      sender_thread.start()
98      reciever_thread.start()
99      sender_thread.join()
100     print(data_dict)
101     ans = ""
102     for i in range(requests):
103         ans += data_dict[i]
104     writeToFile(ans, 'output_thread.txt')
105     md5_hash = hashlib.md5()
106     md5_hash.update(ans.encode('utf-8'))
107     md5_hex = md5_hash.hexdigest()
108     # print(md5_hex)
109     submit_command = f'Submit: cs1210552@bots\nMD5: {md5_hex}\n\n'
110     sock.sendto(submit_command.encode(), server_address)
111     data, server = sock.recvfrom(2096)
112     data = data.decode()
113     print(data)
114     finish = time.time()
```

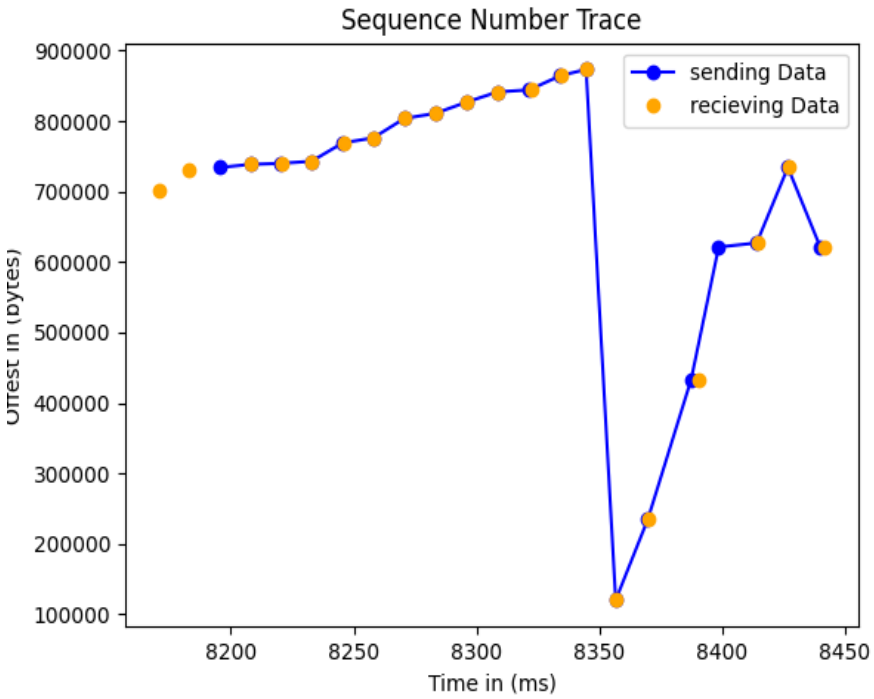
§3 Graphs

§3.1 Sequence-number trace

Here is the graph for sequence-number trace, here all requests are sent parallely using threads, also orange is for receiving the request and blue is for sending the request. Here for this graph we used big.text with max lines of 10000. We can easily see the implemetation using parallel threading in the graphs.

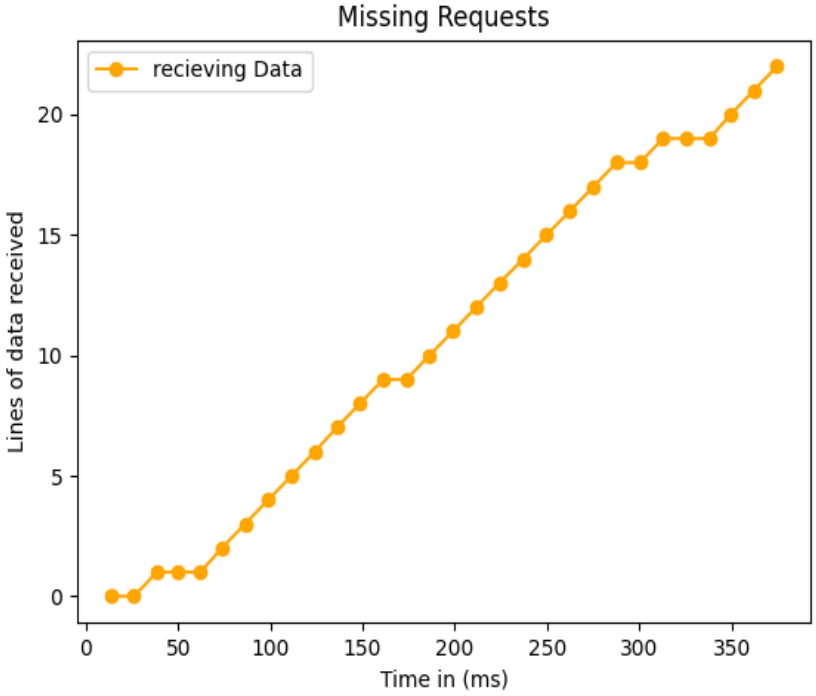






§3.2 Packets received vs Time

We observe that the lines are horizontal when we have sent the request for the offset but that haven't increased the size of data that we currently have. In other words we have not received the data corresponding to those requests sent.



§3.3 Bytes vs Time

Here is bytes vs time analysis, we observed how time increases as we increase the bytes of data. We observed the following curve, where we observed that time is directly proportional to bytes taken; they vary linearly with each other.

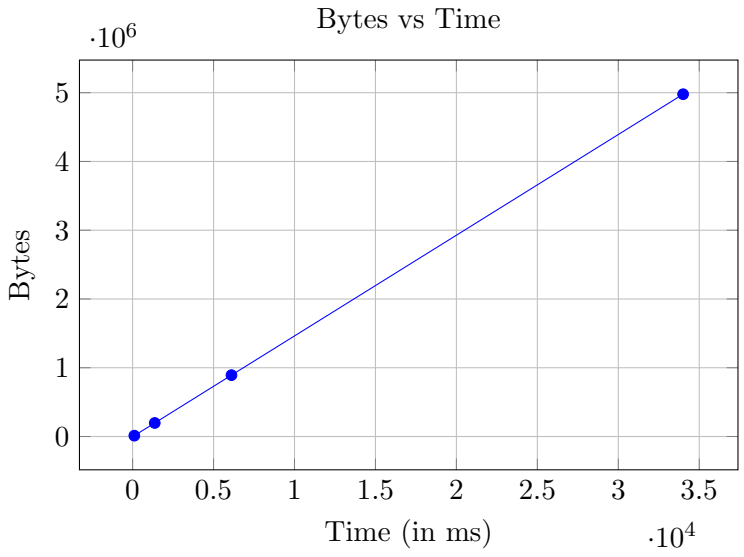


Figure 1: Bytes vs Time

§3.4 Lines vs Time

Here is the Lines vs Time analysis, where we observed how time increases with the number of lines. We can see that time and the number of lines are positively correlated but not directly related. This followed the expected outcome.

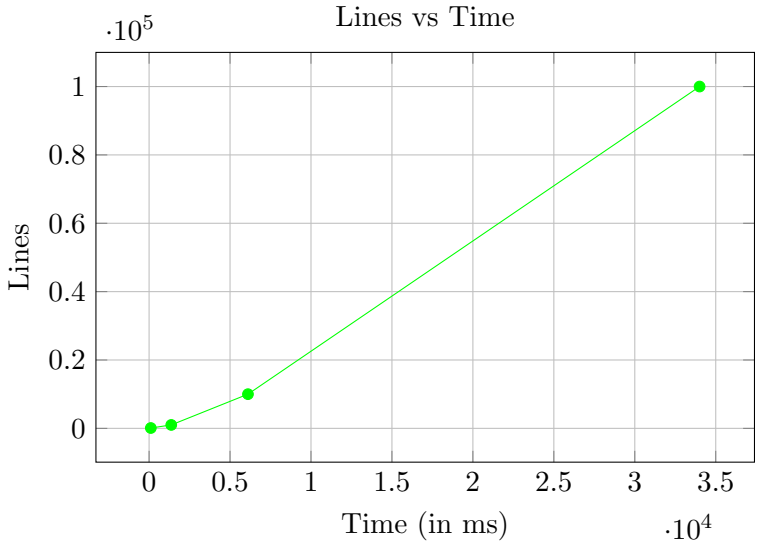


Figure 2: Lines vs Time