Computer Networks

Assignment3 : Milestone2

by Mani Sarthak & Harshit Gupta

October 25, 2023

Some points to be considered:

- All the codes are managed on **GitHub** and can also be found in this **folder**.
- Report of milestone 1 can be found **here**.
- For milestone 2 we had applied the following strategies to get the data.
  - 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.
- **Parameter tuning** for optimisation have been tried out.
- Large data graphs were plotted using **matplotlib** library and rest using **LaTeX**.

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

Contents

1 Logic Overview **3**

2 Experiments **3**

3 Code overview **4**
3.1 Implementation 4

4 Data and Results **6**

5 improvement in Time **7**

6 Analysis **7**
6.1 Sequence-number trace 7
6.2 Congestion Window vs Time 8
6.3 Bursts vs Time 11
6.4 Squish vs Time 12
6.5 Duplicates vs Time 13
6.6 Efficiency vs Time 14
6.7 Bytes vs Time 15
6.8 Lines vs Time 15

§1 Logic Overview

Here the algorithm and logic used is **AIMD** congestion control algorithm. The parts for sending the request, receiving the response, appending the data, submitting the data, hashing it are all the same as part 1 so no changes are done here. Also here threading is used with separate threads for sending and receiving the data. Major changes are done in how we are sending the data compared to before. In part1 we send the data as bursts throughout the implementation and parallely receive it. But in part2 we applied **AIMD** , so we maintained a congestion window, so each time we send requests as a burst of length = congestion window, also between each burst we have a waiting time to let the server recover some **tokens** used. As per **AIMD** we change the congestion window as per follows:-

- if we successfully receive more than 90% of requests we send in a burst then we increase the congestion window by 1
- if we successfully receive less than 90% of requests we send in a burst then we halve the congestion window .

§2 Experiments

1. **Experiment 1**
   - in first experiment with **AIMD** we used the aimd algorithm as given in the book, we made **2 Phases** here.
   - in **Phase 1** it is slow start so basically we start slow so we increase the **cnwd**(congestion window) for each successful response and for atleast a single time out in a burst we halve the **cnwd**. we move to phase 2 when our cnwd $\geq$ threshold, which we fine tune ourselves.
   - in **Phase 2**, it is congestion avoidance, here we increase the congestion window by 1/cnwd for each successful response in a burst and for atleast one timeout we move back to **Phase 1** with cwnd =1 and threshold halved.
2. **Experiment 2**
   - here in this second experiment it is **AIMD** with basic principles such as halving when we have a less than 90% successful responses and increasing by 1 otherwise
   - only difference here is that we limit the max cnwd rather than having it increase unconditionally, so we define a **max_congestion** variable such that cwnd $\leq$ max_congestion every time. we used it on the basis of bucket size
3. **Experiment 3**
   - here in this third experiment it is **AIMD** which is same as second experiment
   - only difference here is that we do not limit the max cnwd, we let it increase unconditionally so we can send more requests in a burst than second one

In practice the best results we get was from Experiment 2 followed by Experiment 3 but since we don't get the bucket size from vayu server and if we try to guess it initially we get squished so we adopted experiment 3 with suitable parameters so as to not get squished and the get the best possible time.

§3 Code overview

§3.1 Implementation

1. **Initialization**:
   - The server's address and various constants are set.
   - Helper functions for file writing, data extraction, and size parsing are defined
   - Parameters such as timeout and waiting_time which we fine tuned with various experimenation
2. **Receiving and Sending Threads**:
   - Two threads are created to handle **AIMD** 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. it works for till all requests in a congestion window are done then it closes. It wait for time = timeout for each request in the burst and then it updates **cnwd** based on it
   - `sending_thread` sends requests to the server based on a list of offsets and sizes, it sends requests in a burst size = congestion window and then create the recieving thread and wait for it to finish and then wait for time = waiting_time
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.
   - `sender_thread` is started which we wait to finish.
   - After `sender_thread` is 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.

Below is the picture of code segement to show the code.

```python
def extract_data(input_text):
    parts = input_text.split("\n\n", 1)
    return parts[1]

def recieving_thread1(file_size,requests,arr):…

def sending_thread(file_size,requests,arr):…

def writeToFile(data, file):
    with open(file, 'w') as f:
        f.write(data)
    f.close()

def main():
    # s=time.time()

    file_size = getSize()
    # buffer=bytearray(file_size)
    arr = [[x, min(x+maxSize, file_size)-x] for x in range(0, file_size, maxSize)]
    requests = len(arr)
    sender_thread = threading.Thread(target=sending_thread, args=(file_size,requests,arr))
    sender_thread.start()
    sender_thread.join()
    ans = ""
    for i in range(requests):
        ans += buffer_dict[i]
    writeToFile(ans, 'output_thread_amid.txt')
    md5_hash = hashlib.md5()
    md5_hash.update(ans.encode('utf-8'))
    md5_hex = md5_hash.hexdigest()
    print(md5_hex)
    submit_command = f'Submit: cs1210552@bots\nMD5: {md5_hex}\n\n'
    sock.settimeout(1)
    sock.sendto(submit_command.encode(), server_address)
    f=time.time()
    print(f'Time taken: {f-s}')
```

This is the implementation of main() function which have the sender and receiver threads.

§4 Data and Results

1. **For code we adopted i.e Experiment 3**
   - **Parameters**
     a) waiting_time - 0.0035 sec
     b) timeout - 0.021 sec
   - **Results**
     a) Server and port:- 10.17.7.134 , 9801
     b) Requested Packets:-2725
     c) Recieved Packets:- 2294
     d) Missed Packets:-451
     e) Time taken:- 19.8954 sec
     f) Hash generated:- ae54625f5680d2cf03224ebf23be481e
     g) Duplicate packets:- 12
     h) Penalties:-81
     i) Squished:- NO
2. **For Experiment 2**
   - **Parameters**
     a) waiting_time - 0.03 sec
     b) timeout - 0.0027 sec
   - **Results**
     a) Server and port:-10.17.7.134 , 9801
     b) Requested Packets:- 2643
     c) Recieved Packets:- 2280
     d) Missed Packets:- 363
     e) Time taken:- 14.9255 sec
     f) Hash generated:- ae54625f5680d2cf03224ebf23be481e
     g) Duplicate packets:- 16
     h) Penalties:-71
     i) Squished:- NO

§5 improvement in Time

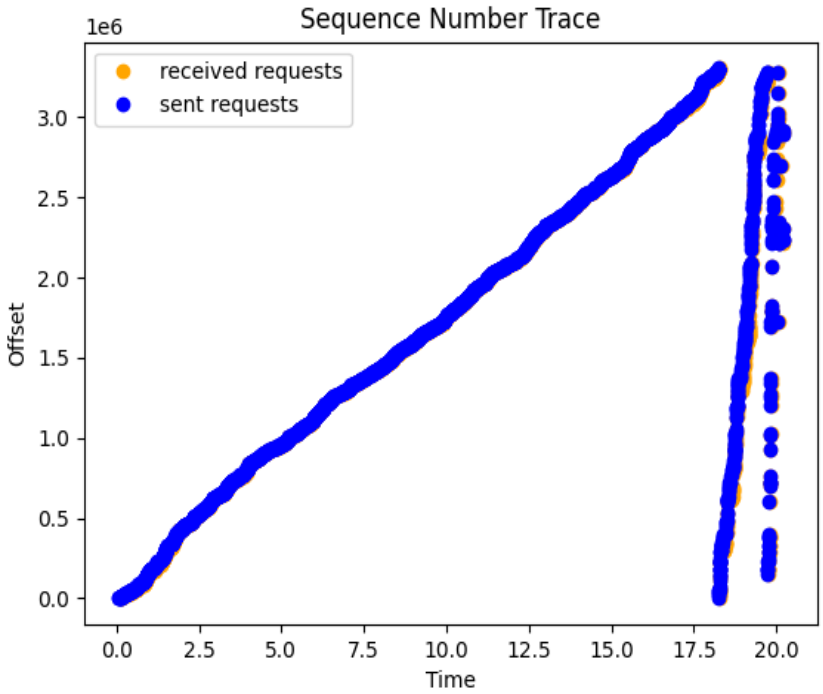
Compared to previous part 1 there is significant time improvement as shown :-

Without AIMD	With AIMD
86.883 sec	19.8954 sec

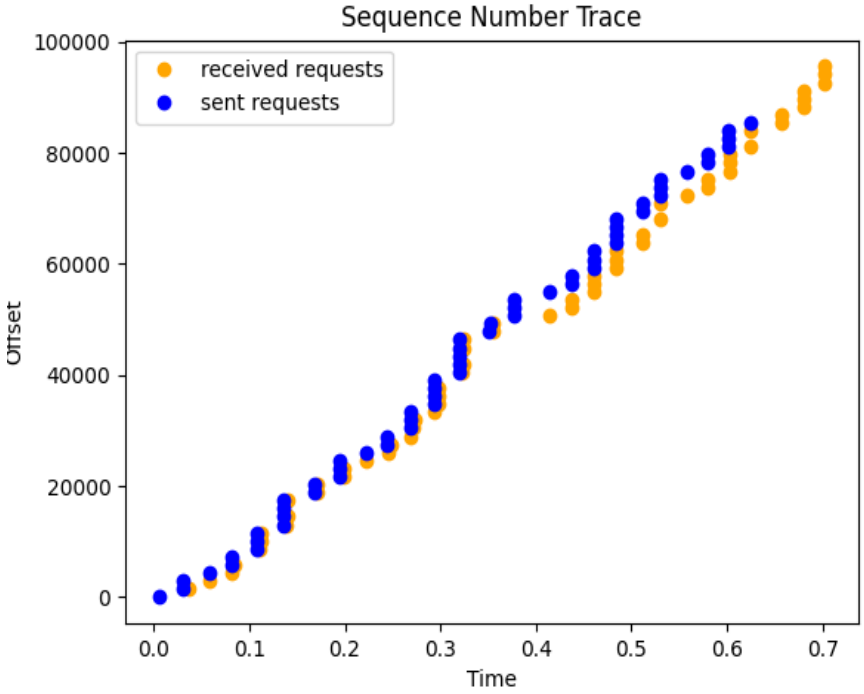
§6 Analysis

§6.1 Sequence-number trace

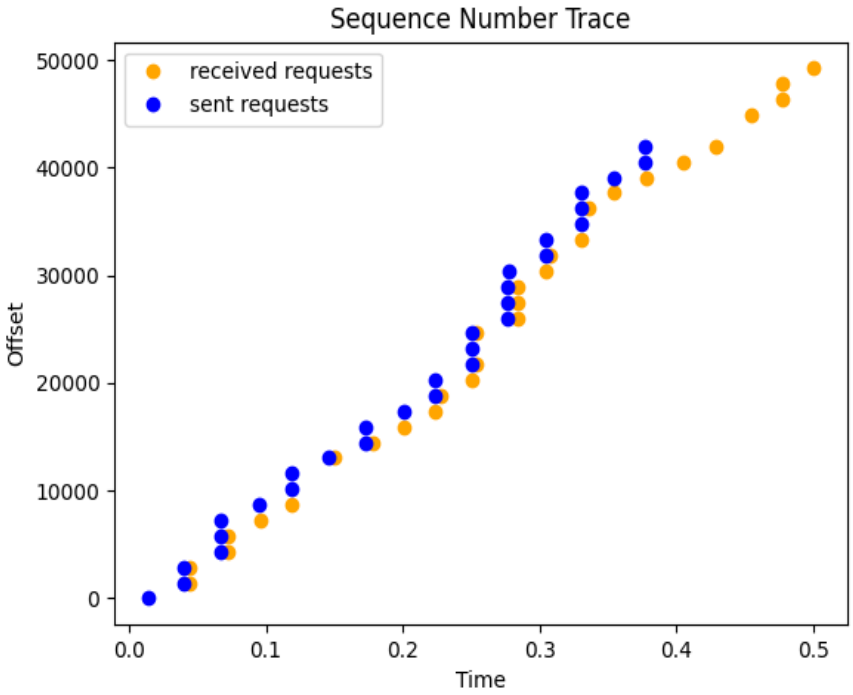
Here is the graph for sequence-number trace also orange is for receiving the request and blue is for sending the request. We can easily see the implementation of AIMD in the images. We observe that the requests and data are being sent and received in bursts.



This is the graph for complete time, here vayu server is 10.17.7.134, time is in seconds.



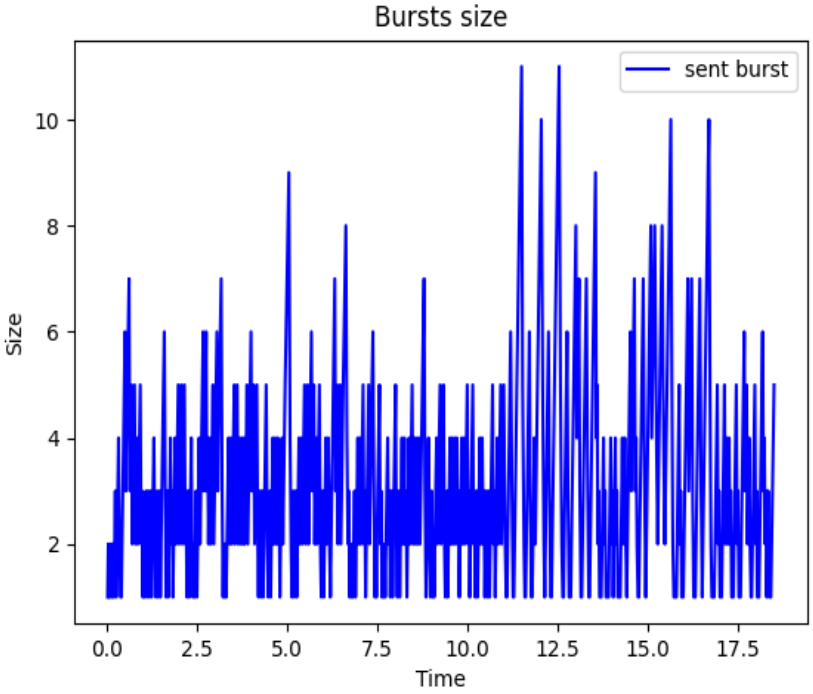
This is zoomed out version of above graph, time in seconds.



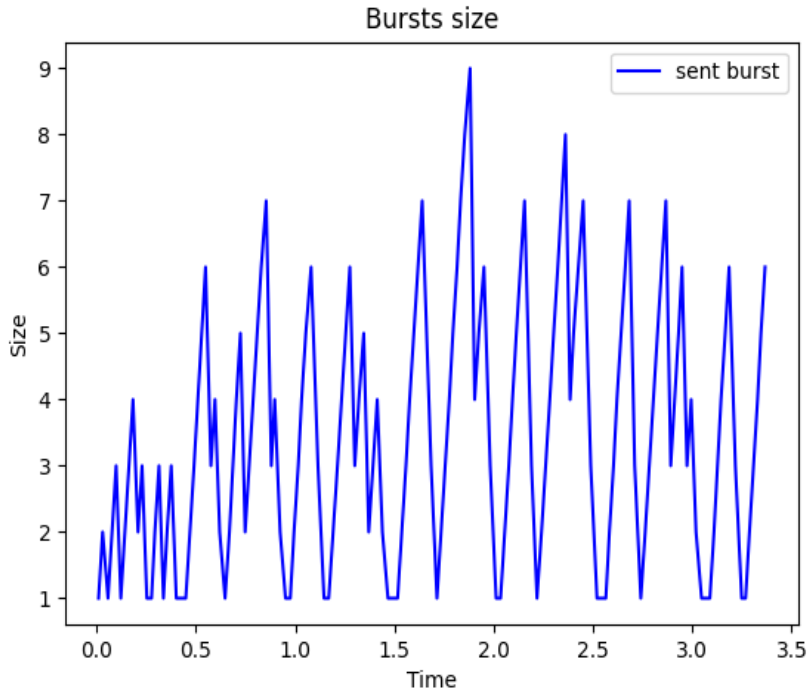
This is hyper zoomed out version of above graph

§6.2 Congestion Window vs Time

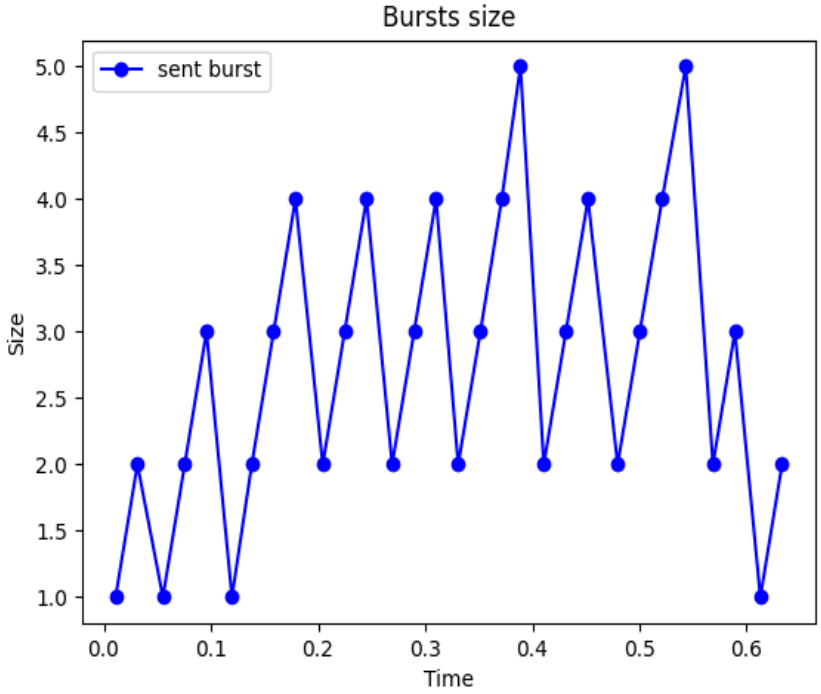
This section is to visualise how the burst size vary with time , in AIMD most important parameter is burst size which we control. In these graphs, **saw-tooth** behaviour can be clearly seen which is a basic characteristic of AIMD



This is burst size/congestion window vs time for full time



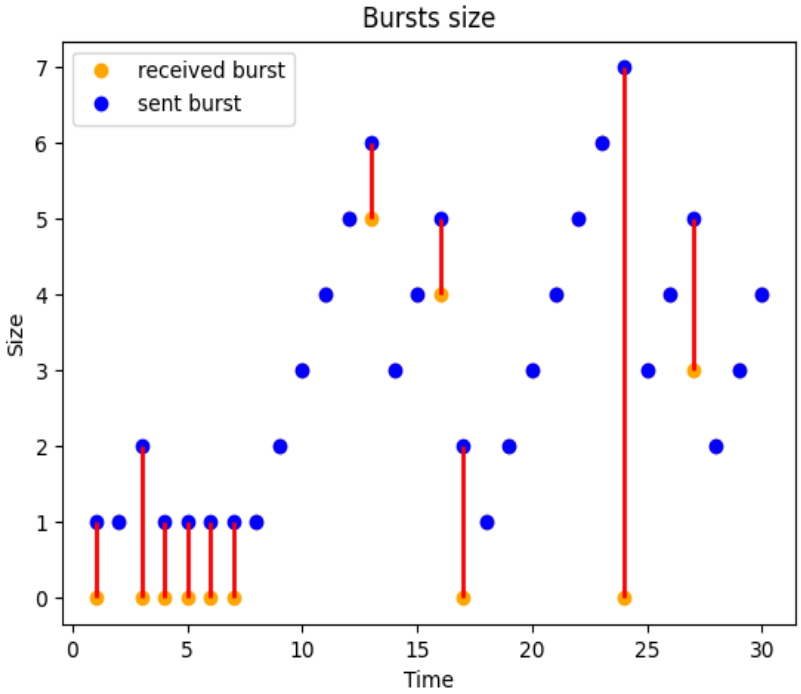
this is zoomed out version of above graph



This is more zoomed out version of above graph but here how aimd behaves can be clearly seen, we can see how burst size vary, it increased by 1 and decrease by half

§6.3 Bursts vs Time

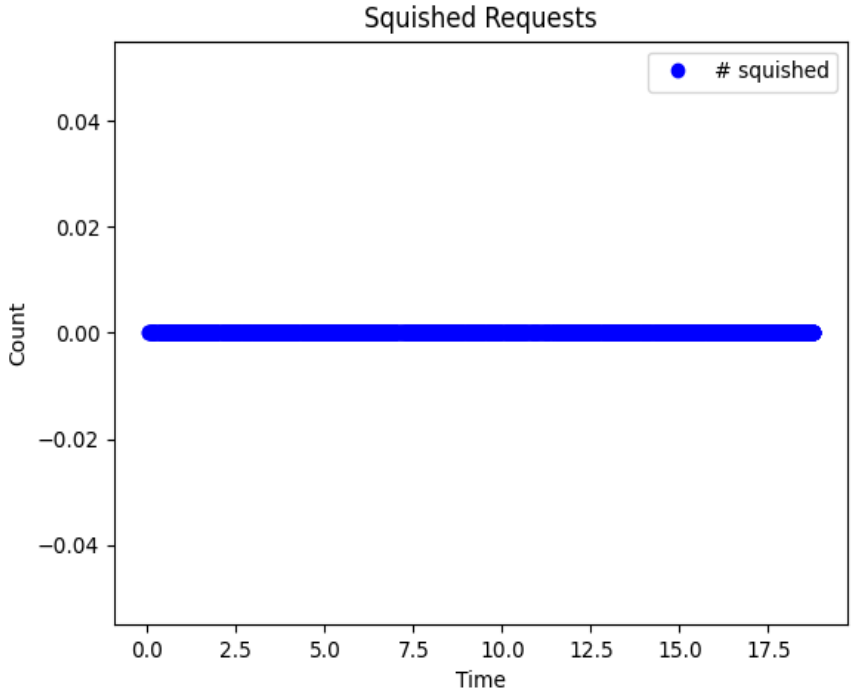
This section shows the stats for the size of bursts in sending and receiving process. When there is a single point it means all the data have been received for the sent requests. Observe how the bursts size for in requests increase linearly. Also there is no fixed relationship between how many bursts are sent and how many are received (clearly shows the RTT is varying, since the requests may be coming after the waiting window have passed).



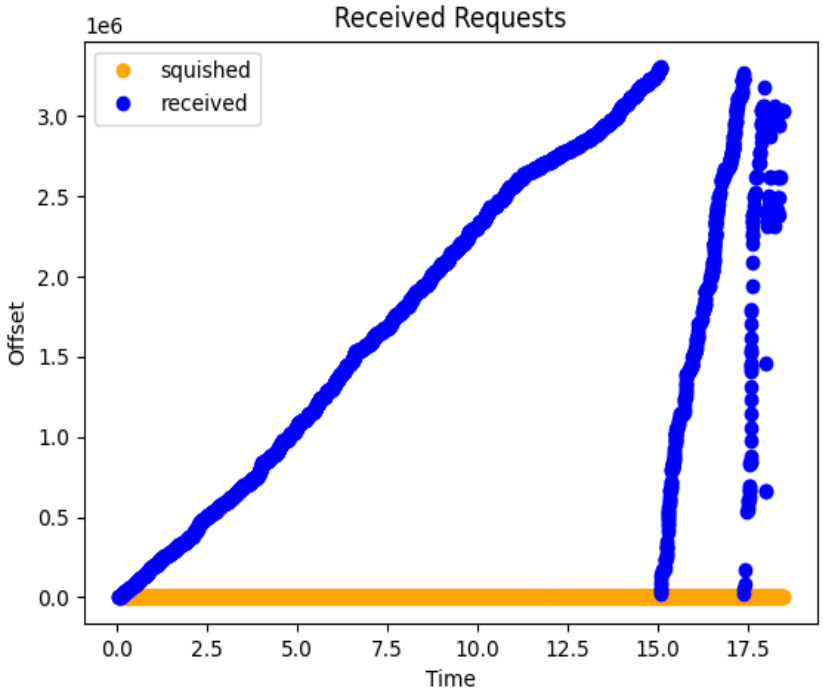
This is size of bursts(sent and received) vs time.

§6.4 Squish vs Time

This section is to visualise how the squishing vary with time. This is to show how many squish we get with our code in the time. here our implementation we get 0 squishes. Hence our code is working as expected.



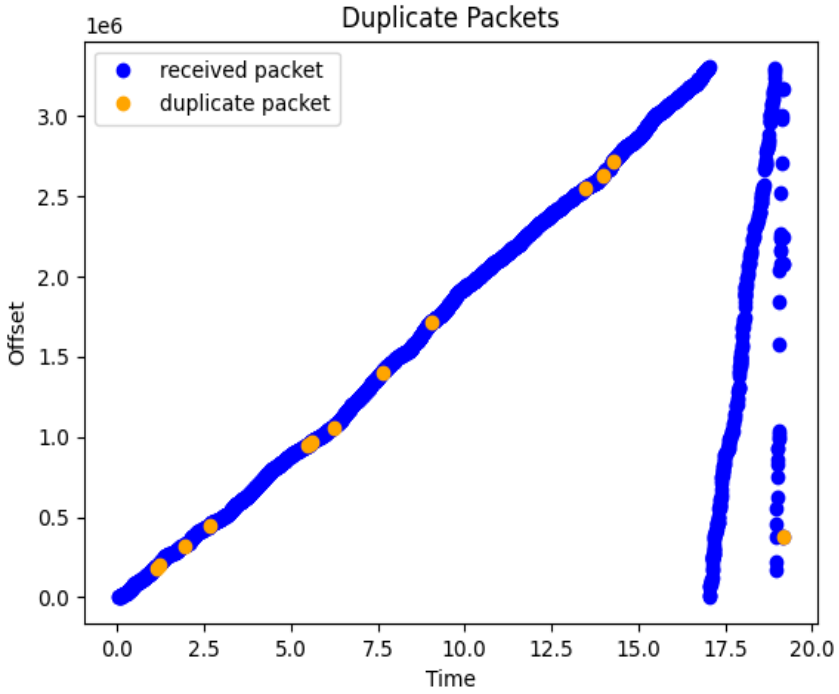
This is number of squishes vs time for full time



This is to visualise how squish vary with recieved request and time(in seconds).

§6.5 Duplicates vs Time

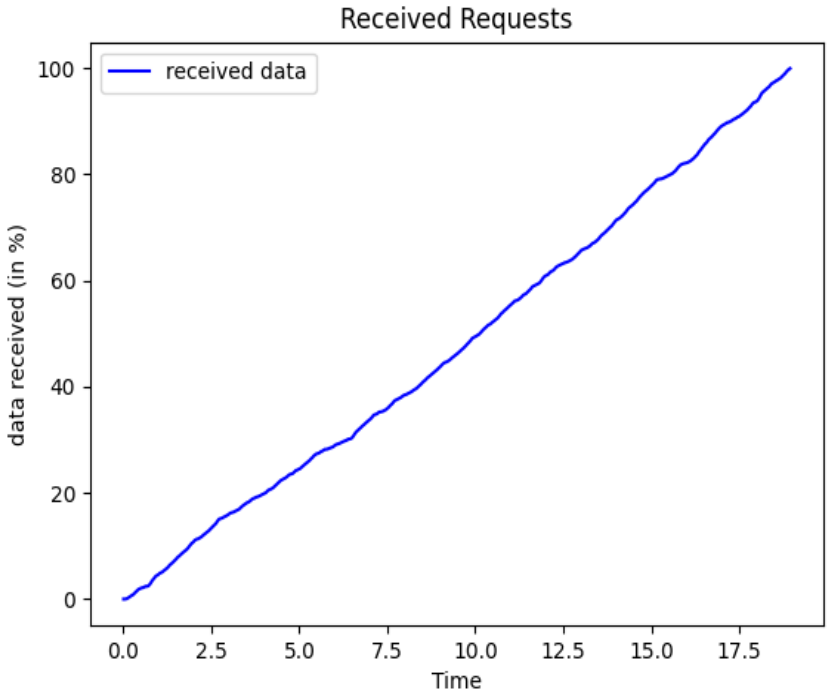
This section is to visualise how the duplicates vary with time. This section is to show how duplicate responses we get over time.



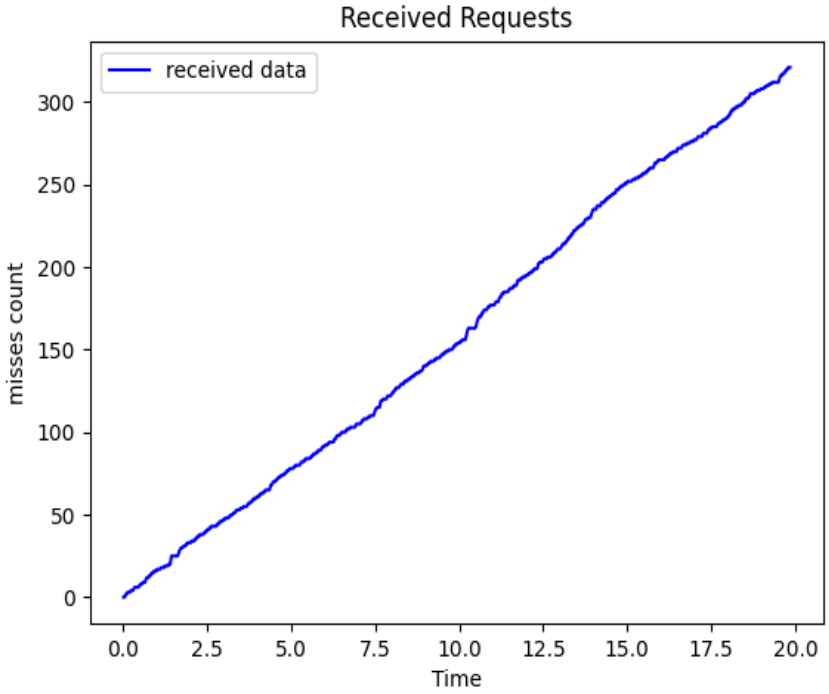
This is duplicates we get over responses recieved vs time

§6.6 Efficiency vs Time

This section shows the relationship between missed requests and fraction of data fetched versus time.



This is to visualise the percentage of data received w.r.t time.



This is to visualise how many misses have been there on with the requests sent

§6.7 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.

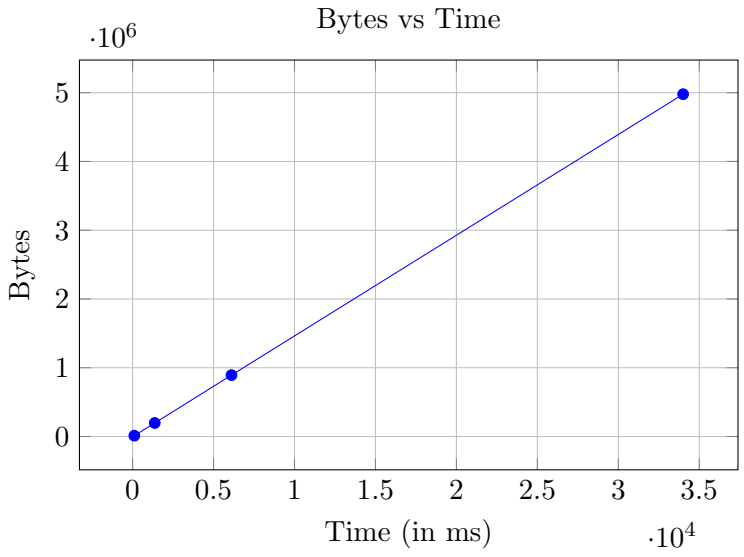


Figure 1: Bytes vs Time

§6.8 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.

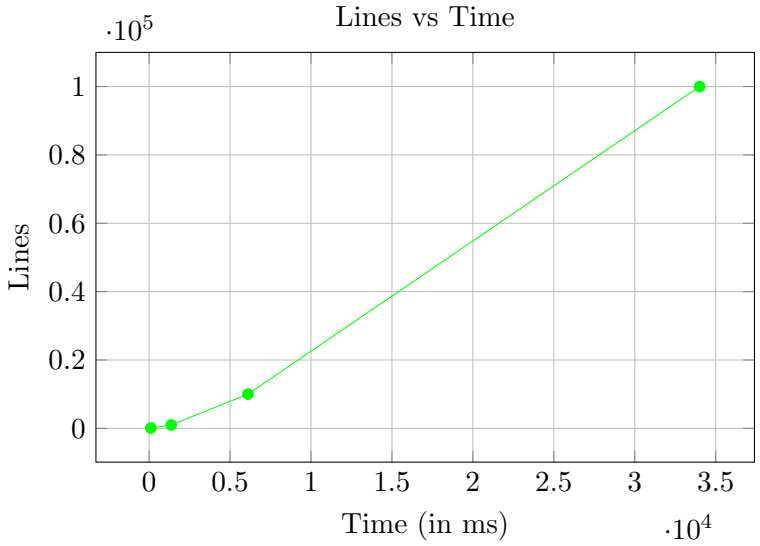


Figure 2: Lines vs Time