Assignment 3 : Receiving data reliably

Rajat Golechha, Satyam Kumar

November 1, 2023

Contents

1 What We Did

In this milestone of the assignment, we implemented reliable data transfer with adequate rate without getting squished using the UDP protocol for server with variable rate for leaky bucket parameters.

1.1 How We Did It

First, we requested the server for the size of the file to be obtained. Based on the file size, we created a list of requests that needed to be sent to the server over a period of time. Then we made a list of Receive time for first 30 requests. Then we sorted it and find the median of list and declare it as our Receive time, and use it as a parameter for deciding all sorts of waiting times like socket timeout, wait between bursts, etc.

```
    rtt_l = []
for i in range(30):
    s = time.time()
    if(len(reqs) == 0):
        break
    x = i % len(reqs)
    req_msg = "Offset: "+str(reqs[x][0])+"\n"+"NumBytes: "+str(reqs[x
    ][1])+"\n\n"
    UDPClientSocket.sendto(req_msg.encode(),serverAddressPort)
    try:
        msgFromServer = UDPClientSocket.recvfrom(bufferSize)
        msg = msgFromServer[0]
        e = time.time()
        msg = msg.decode("utf-8")
        x = msg.find('Offset')
        y = msg.find('NumBytes')
        offset = int(msg[x+8:y-1])
        m1 = msg[y+9::]
        z = m1.find('\n')
        num_bytes = int(m1[0:z])
        m2 = m1[z+2:z+2+num_bytes]
        datadict[offset] = m2
        element = [offset, num_bytes]
        if element in reqs:
            reqs.remove(element)
        rtt_l.append(e-s)
        time.sleep(0.015)
    except socket.timeout:
        continue
rtt_l.sort()
x = len(rtt_l)
mid = x//2
rtt = (rtt_l[mid] + rtt_l[~mid])/2
if rtt < 0.004:
    rtt = 0.004
print("RTT is :",rtt)
UDPClientSocket.settimeout(rtt*2.5)
```

1.2 Sending and Receiving Offsets

In the sending part, we follow a strategy similar to AIMD but with EWMA for Receive time. We have initial burst_size which is equal to 5. Then we are changing burst size dynamically based on how much fraction of requests has come. We have outer while loop which is

running until all requests have come. Then we start timer and send a burst of offsets equal to burst_size without any gap. Then we intialize a resp counter which keep the store of how many replies come from server in context of above requests.

We also have empty list to store round trip time for packets that has come in order to calculate Receive time using Exponentially weighted moving approach. We have ignored those packets which have experienced time out or are already present in the buffer and in order to know this we have variable called rtt_cal. Now we have a for loop of size equal to burst size+1. We try to receive message from server and if reply does not come before time_out which is rtt*2.5 we move to next iteration of the loop. If reply comes then we note its time and store it in a list. We use all the elements stored in the list and compute their median.

We use the median to estimate a new estimated_rtt (RECEIVE TIME) and then update the old receive time with a learning rate using the formula 0.7*rtt+0.3*(new estimated_rtt). We also have an upper bound on receive time to prevent it from growing too much due to one false positive and similarly a lower bound to prevent it from falling too fast.

We also update timeout of ClientSocket by setting it to new rtt*2.5. After that we have a sleep period so that we are not get deprived of tokens. Sleep period is (burst size+2)*rtt*0.5. Now once we have the number of responses, we like AIMD check if all the messages are received or not, if all of the messages are received and we do not get squished then we increment the burst-size by 1, else we halve the sending rate for the messages exactly like AIMD.

```
UDPClientSocket.settimeout(rtt*2.5)
# now we have the RTT and the reqs to send, so let's start sending the
    messages using AIMD
burst_size = 5
i = 0
while(len(reqs) > 0):
    if(len(reqs) <= burst_size):
        burst_size = len(reqs)
    print("Burst size is %d" % burst_size)
    print(len(reqs))
    sendtime = time.time()
    for j in range(burst_size):
        x = i % len(reqs)
        i = i + 1
        req_msg = "Offset: "+str(reqs[x][0])+"\n"+"NumBytes: "+str(reqs[x
    ][1])+"\n\n"
        UDPClientSocket.sendto(req_msg.encode(),serverAddressPort)
    resp = 0
    squished = False
    L = []
    rtt_cal = False
    for j in range(burst_size+1):
        try:
            msg = UDPClientSocket.recvfrom(bufferSize)
            msg = msg[0]
            msg = msg.decode("utf-8")
            x = msg.find('Offset')
            y = msg.find('NumBytes')
            offset = int(msg[x+8:y-1])
            m1 = msg[y+9::]
            z = m1.find('\n')
            num_bytes = int(m1[0:z])
            if (m1[z+1:z+10]=='Squished\n'):
                squished = True
                print("S    Q    U    I    S    H    E    D")
```

```
                m1 = m1[0:z+1] + m1[z+10:]
            m2 = m1[z+2:z+2+num_bytes]
            datadict[offset] = m2
            element = [offset, num_bytes]
            if element in reqs:
                reqs.remove(element)
            resp = resp + 1
            if(rtt_cal == False):
                recv_time = time.time()
                rtt_1 = recv_time - sendtime
                if(rtt_1 < 0.004):
                    pass
                else:
                    L.append(rtt_1)
        except socket.timeout:
            rtt_cal = True
            continue
    if(len(L) > 0):
        L.sort()
        x = len(L)
        mid = x//2
        rtt_ = (L[mid] + L[~mid])/2
        rtt = 0.7*rtt + 0.3*rtt_
        if(rtt < 0.004):
            rtt = 0.004
        if(rtt > 0.01):
            rtt = 0.01
        print("RTT is :",rtt)
        UDPClientSocket.settimeout(rtt*2.5)
    time.sleep((burst_size+2)*rtt*0.5)
    print(resp,burst_size)
    if (resp >= burst_size and squished == False):
        burst_size = burst_size + 1
    else:
        burst_size = (burst_size + 3) // 2
```

1.3 Submission on Server

Once all the messages in the request list were received, we concatenated them and generated an MD5 hash for the entire message. We then keep on submitting the submit message until we receive a Result from the server end, we do this in order to prevent any unforseen packet drop at the last moment, and establish a reliable final data transfer.

```
    submit = ""
    for i in range(num_packets):
        submit += datadict[1448*i]

    md5_hash = hashlib.md5(submit.encode('utf-8'))
    md5_hex = md5_hash.hexdigest()
    print(md5_hex)
    submit_cmd = "Submit: Doof\n" + "MD5: " + md5_hex + "\n\n"
    while True:
        try:
            UDPClientSocket.sendto(submit_cmd.encode(), serverAddressPort
    )
            msg = UDPClientSocket.recvfrom(bufferSize)
            msg = msg[0].decode()
            if(msg[0:6]=='Result'):
                break
```

```
        except socket.timeout:
            continue
    print(msg)
    UDPClientSocket.close()
```

2 Graphs

On analysing the data and converting them to graphs, we obtain the following plots :

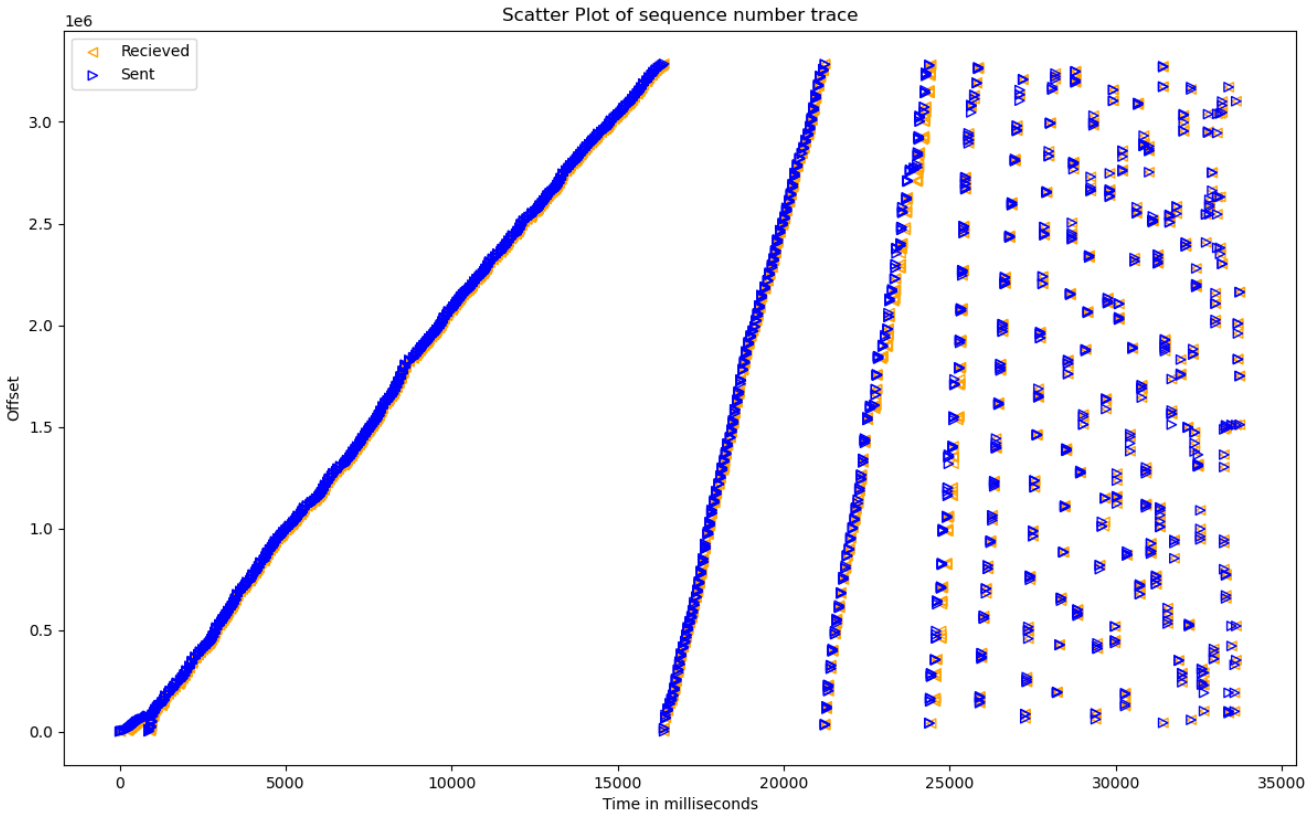

Figure 1: Offset-Time plot

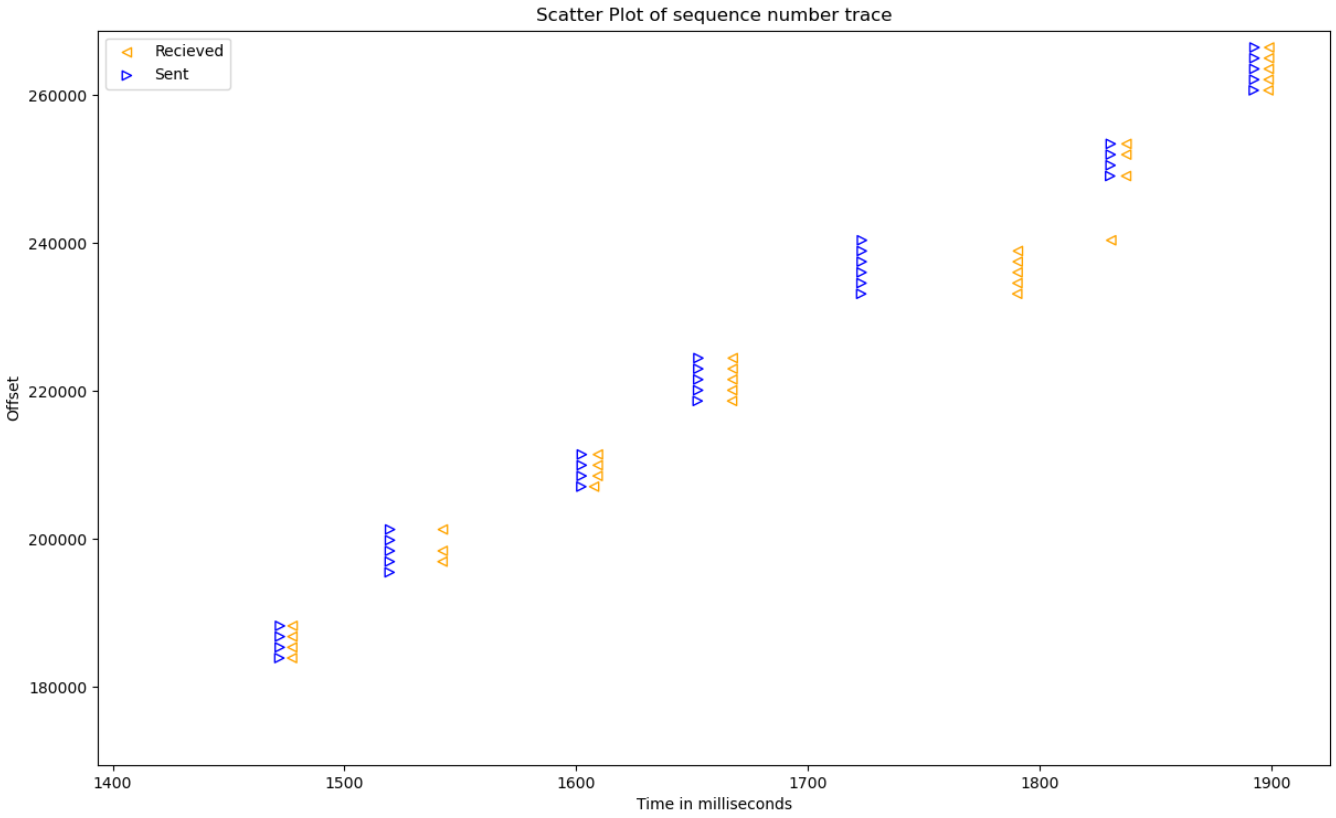

Figure 2: Zoomed in Offset-Time plot

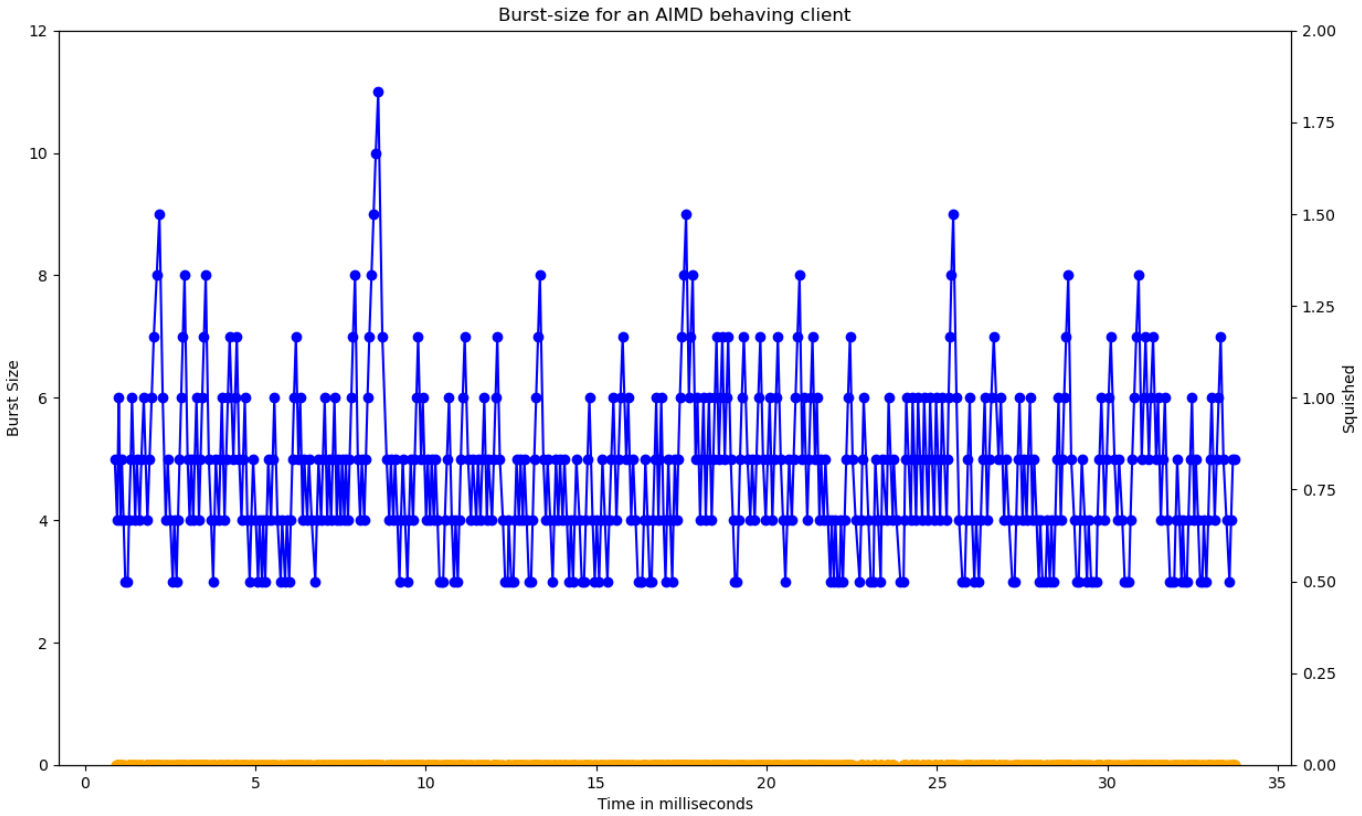

Figure 3: Burst-size in a variable rate server

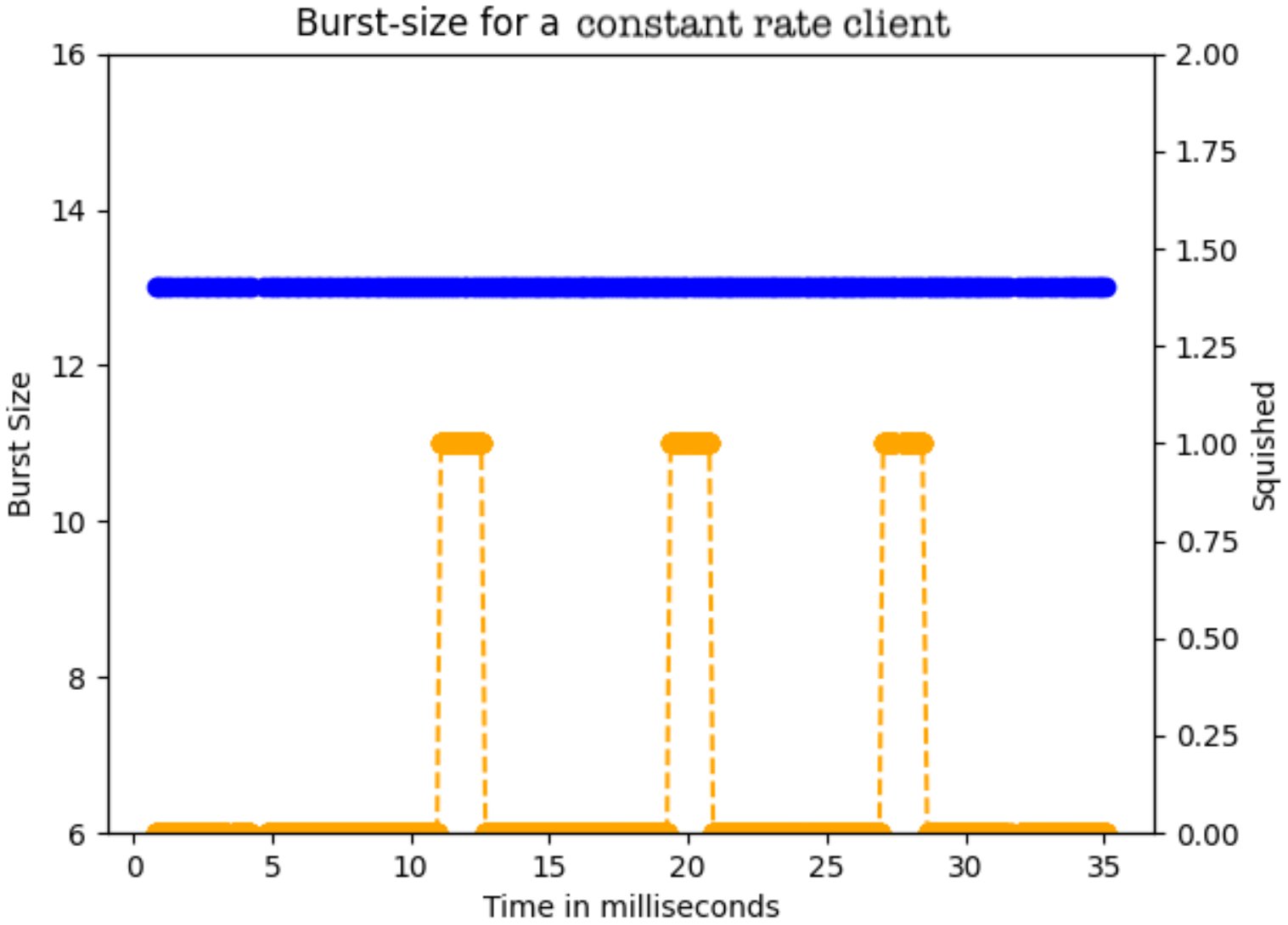

Figure 4: Burst-size in a variable rate server

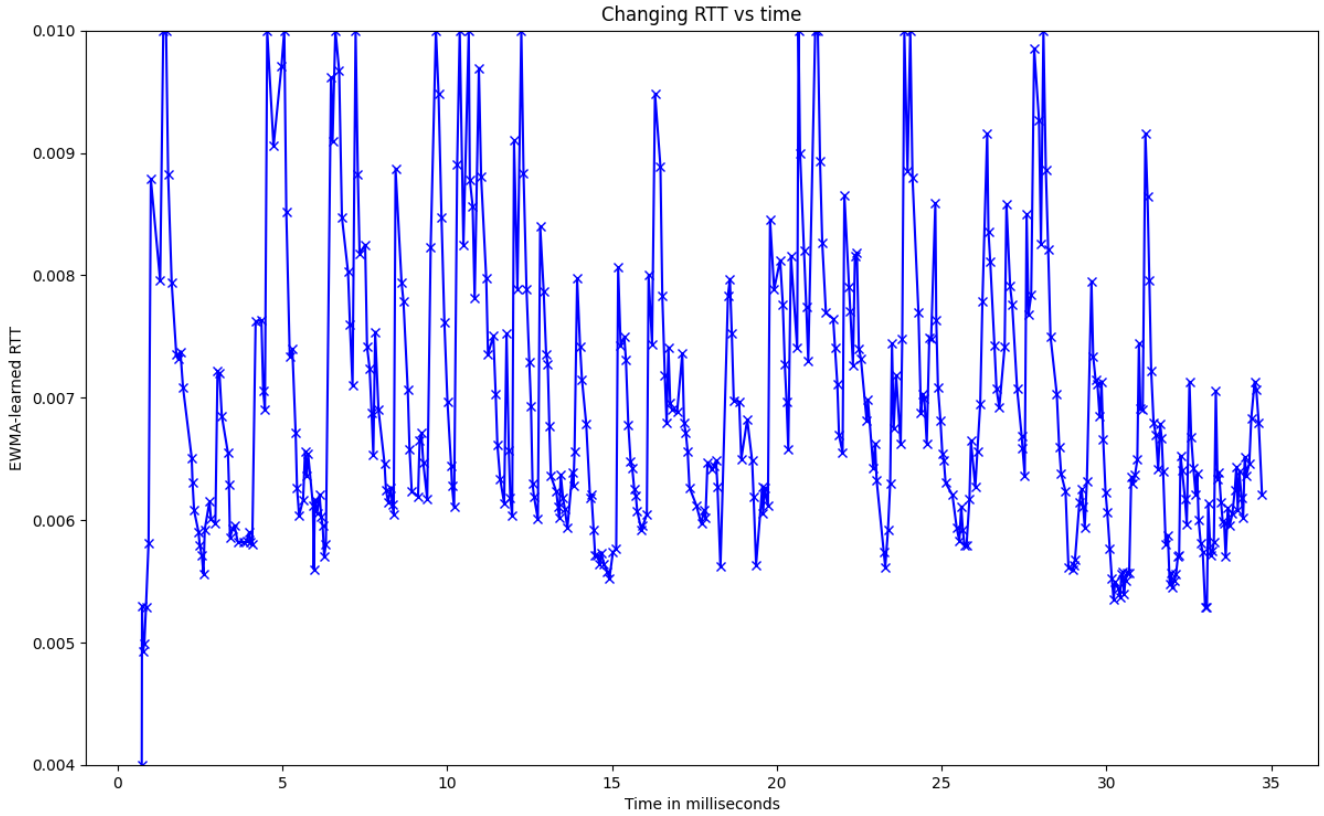

Figure 5: Changing Receiving Time with time

3 Runtime Data

Time Taken	Penalty Incurred
32.236 sec	26
33.358 sec	18
30.179 sec	3
31.238 sec	7

Table 1: The following data is for vayu.iitd.ac.in

Time Taken	Penalty Incurred
24.659 sec	16
23.138 sec	12
23.949 sec	16
22.246 sec	18

Table 2: The following data is for localhost server

4 Graph Analysis

4.1 Sequence number trace

- In the plots, the blue mark represents the sending of a request for a packet, and the orange mark represents the receiving of a packet by the receive thread.
- In the first graph, we can observe multiple straight lines, each representing different iterations of requests. Every subsequent line corresponds to packets that were either dropped, skipped, or lost in the previous iterations.
- To ensure the reliability of transfer, multiple iterations are performed. This iterative process guarantees that every packet is received, addressing any potential issues associated with packet loss.
- The client is programmed to run through the entire request space in a monotonically increasing order, then make another pass to request for offsets that were not received in the first round, then another, and so on, until all offsets are received
- We also observe in the second zoomed in graph how our burst size changes with time. Initially we have sent a burst of 4 requests and all 4 requests has come so we increment burst by 1. Then again we get all requests back so we again increment burst by 1. But we get a packet loss of 1 so we do (burst+3)/2.

4.2 Burst Size vs time

- Figure 3 shows the burst-size trace for our client. The client is able to successfully adjust to larger burst-sizes during periods when the server allows a higher rate, and reduces to lower burst sizes during periods of lower rate. The client does not get squished even once and follows the sawtooth pattern expected of an AIMD client.
- On the other hand, Figure 4 shows a burst-size trace for our trivial client implementation that maintains a constant burst-size. Here, the client is able to recover from the squish period periodically especially when it coincides with a higher rate period at the server when new tokens are generated more quickly, but gets squished soon after when the server reduces the rate.
- Our client is able to beat the constant burst-size client in such variable rate settings because it is able to adjust to the changing available bandwidth. This shows why we can't use a constant rate client against a variable rate server, because if the token replenishment is slower than token exhaustion then the client will be squished for extended periods of time and lead to lots of penalties and maybe even squishing.

4.3 Receive time variation

- We have seen the tradeoff between time and number of penalty we are getting. If we decrease rate of request we get fewer number of penalty but if we increase rate of request we get more penalty and chances of getting squished also become higher. Therefore we appropriately used the socket timeout and sleeps to achieve a decent time with a very low penalty.

- From Figure 5 we can observe that if the server is highly responsive and can process requests swiftly, it results in a lower Receive time. This is because the server can quickly send the response back to the client, reducing the time the client has to wait for the reply. Conversely, when the server's response rate is sluggish, it leads to a higher Receive time. This means that the client must wait longer for the response to arrive, resulting in a longer round-trip time. Our code suitably adjusts the interburst sleep time and socket timeout to match the server's rate.

5 Exception Handling

We handled different kinds of exceptions, including:

- **Packet Drop or Penalty:** In the case of a packet drop, the request is not removed from our list of requests and will be requested again in the next iteration.
- **Skipped Request:** We handle skipped requests in the same manner as packet drops.
- **Submit response drop :** We submit until we receive the Result message through a while loop to handle this error.

6 Conclusion

In a nutshell, the combination of AIMD (Additive Increase, Multiplicative Decrease) and EWMA (Exponentially Weighted Moving Average) proves to be an exceptionally effective strategy when dealing with a variable-rate server. This algorithm demonstrates its prowess by adeptly monitoring the server's response times and keeping a close eye on the number of requests it has successfully handled. By utilizing this dual information, the algorithm adeptly fine-tunes the rate at which it sends requests and strategically incorporates intermittent waiting periods. The primary goal of this approach is to ensure optimal performance without experiencing congestion or incurring a barrage of penalties.