Assignment 3 : Receiving data reliably

Rajat Golechha, Satyam Kumar

October 25, 2023

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 via two threads: one that receives the messages and another that sends the messages according to the protocol.

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 RTT for first 60 requests. Then we sorted it and find the median of list and declare it as our RTT, and use it as a parameter for deciding all sorts of waiting times like socket timeout, wait between bursts, etc. We then initiated the receiving and sending threads.

```
    rtt_l = []
    for i in range(60):
        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]
            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'):
                m1 = m1[0:z+1] + m1[z+10:]
            if(z+2+num_bytes > len(m1)):
                continue
            m2 = m1[z+2:z+2+num_bytes]
            with datadict_lock:
                datadict[offset] = m2
            element = [offset, num_bytes]
            with reqs_lock:
                if element in reqs:
```

```
                    reqs.remove(element)
            t = time.time()
            rtt_l.append(t-s)
            time.sleep(0.015)
        except socket.timeout:
            continue
    # sum_ = 0
    rtt_l.sort()
    x = len(rtt_l)
    mid = x//2
    rtt = (rtt_l[mid] + rtt_l[~mid])/2
    print("RTT is :",rtt)
    if rtt < 0.00005:
        rtt = 0.00005
```

1.1.1 Receiving Thread

The receiving thread continuously concatenated the received messages into a buffer. We parsed the string for complete datagrams. When a complete datagram was found, we removed that part from the string and stored it in a dictionary. Simultaneously, we removed the corresponding element from the request list. We are waiting for 4*rtt seconds for any reply to come, it has been set as the socket timeout and if no reply comes in that window, then the while loop starts once again, this goes on until the reqs are exhausted and once that happens we exit the while loop. And the thread terminates.

```
client.settimeout(4*rtt)
    while True:
        with reqs_lock:
            print(len(reqs))
            if len(reqs) == 0:
                break
        try:
            recv = client.recvfrom(bufferSize)
            recv = recv[0].decode()
            all_data += recv
            while all_data.count('Offset')>0:    .
                x = all_data.find('Offset')
                y = all_data.find('NumBytes')
                offset = int(all_data[x+8:y-1])
                m1 = all_data[y+9::]
                z = m1.find('\n')
                num_bytes = int(m1[0:z])
                if (m1[z+1:z+10]=='Squished\n'):
                    m1 = m1[0:z+1] + m1[z+10:]
                if(z+2+num_bytes > len(m1)):
                    break
                m2 = m1[z+2:z+2+num_bytes]
                with recv_lock:
                    reqs_recv.append(offset)
                with datadict_lock:
                    datadict[offset] = m2
                    all_data = m1[z+2+num_bytes::]
                element = [offset, num_bytes]
                with reqs_lock:
                    if element in reqs:
                        reqs.remove(element)

        except socket.timeout:
            continue
```

1.1.2 Sending Thread

In the sending thread, we follow a strategy similar to AIMD but with a twist. 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 maintain two lists, one in the receiving thread and another in the sending thread, each time we start a new burst we initialise both lists to empty. Then we store the offsets requests in sending list, and offsets received in receiving list, once the burst is over we find the intersection of the two lists. Unlike AIMD our bursts aren't exactly continous rather we use some spacing between the bursts for the tokens to replenish, you can see that we have a window of 0.035 seconds over which our requests are uniformly spaced then we wait for burst-size*rtt*0.2 seconds more to receive any further requests that are in transit. Why these numbers, well the 0.35 second window is an arbitary size and would function the same for any size with suitable changes to other parameters, as for the waiting time for the remaining messages, we tried various parameters like burst-size*rtt, burst-size*rtt*0.5, etc. and we got the best results at around a quarter therefore we chose to stick with it. Now once we have the intersection of these lists, we like AIMD check if all the messages are received or not, if over 90% of the messages are received then we increment the burst-size by 1, if 80-90% of the messages are received then we maintain the same burst size, if there's any more packet loss than 20% then we halve the sending rate for the messages exactly like AIMD.

```
        i = 0
    while True:
        reqs_sent = []
        with recv_lock:
            reqs_recv = []
        for j in range(burst_size):
            with reqs_lock:
                if len(reqs) == 0:
                    break
            with reqs_lock:
                x = i % len(reqs)
                i = i + 1
                req_msg = "Offset: "+str(reqs[x][0])+"\n"+"NumBytes: "+
     str(reqs[x][1])+"\n\n"
                reqs_sent.append(reqs[x][0])
            client.sendto(req_msg.encode(),serverAddressPort)
            y = 0.1/burst_size
            time.sleep(y*0.35)
        time.sleep(burst_size*rtt*0.2)

        with recv_lock:
            L3 = intersection(reqs_sent, reqs_recv)
            x = len(L3)
            if(x >= burst_size * 0.9):
                burst_size = burst_size + 1
            elif(x >= 0.8*burst_size):
                burst_size = burst_size
            else:
                burst_size = (burst_size+3) // 2


        print("burst_size is :",burst_size)
        with reqs_lock:
                if len(reqs) == 0:
                    break
```

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 :

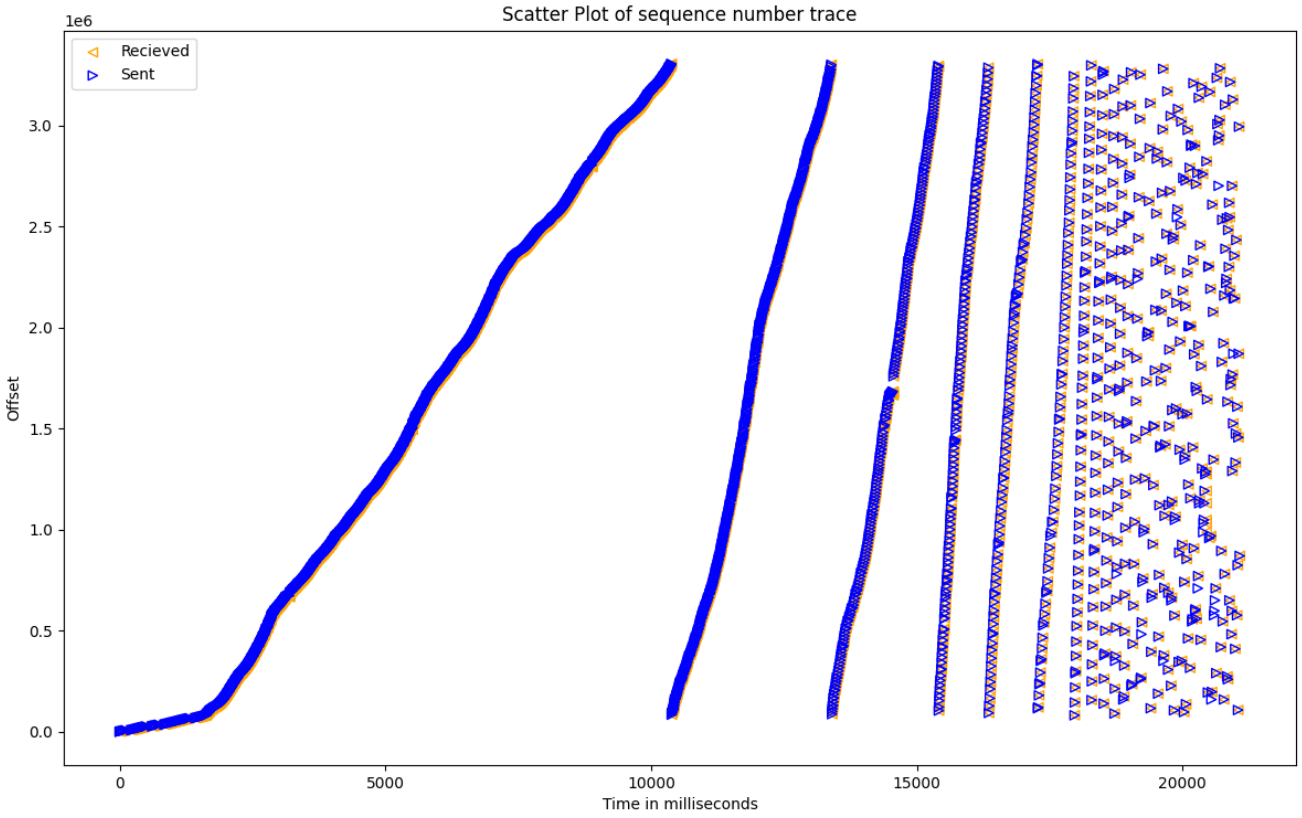

Figure 1: Offset-Time plot

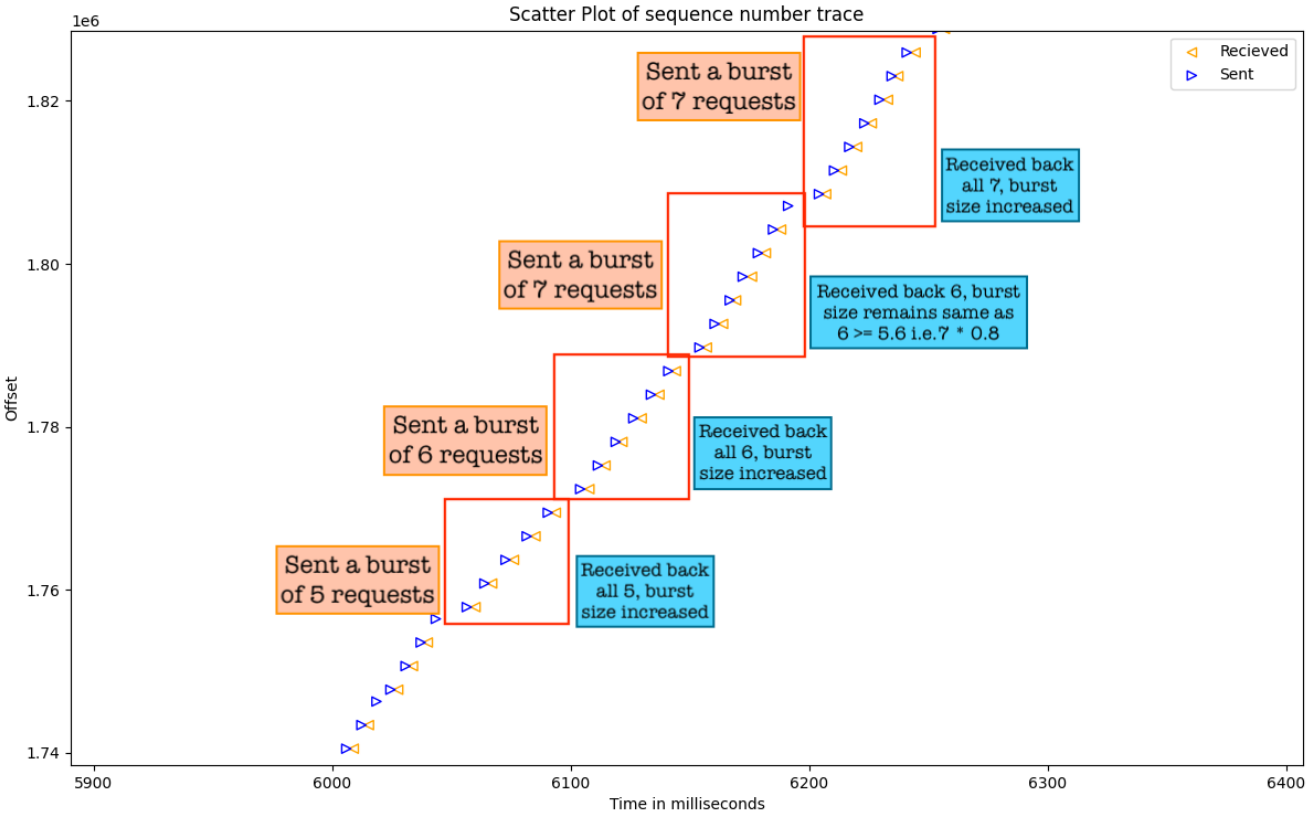

Figure 2: Zoomed in Offset-Time plot

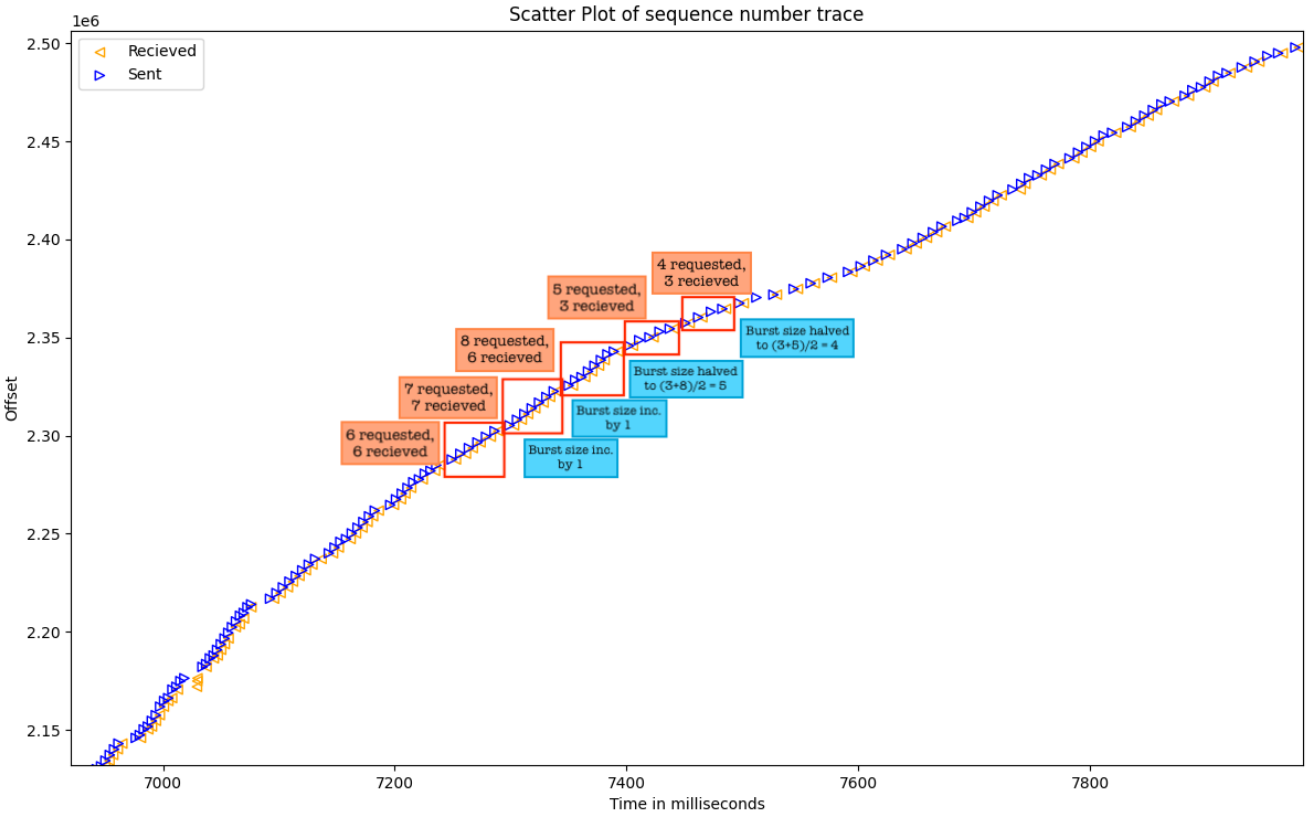

Figure 3: Zoomed in Offset-Time plot

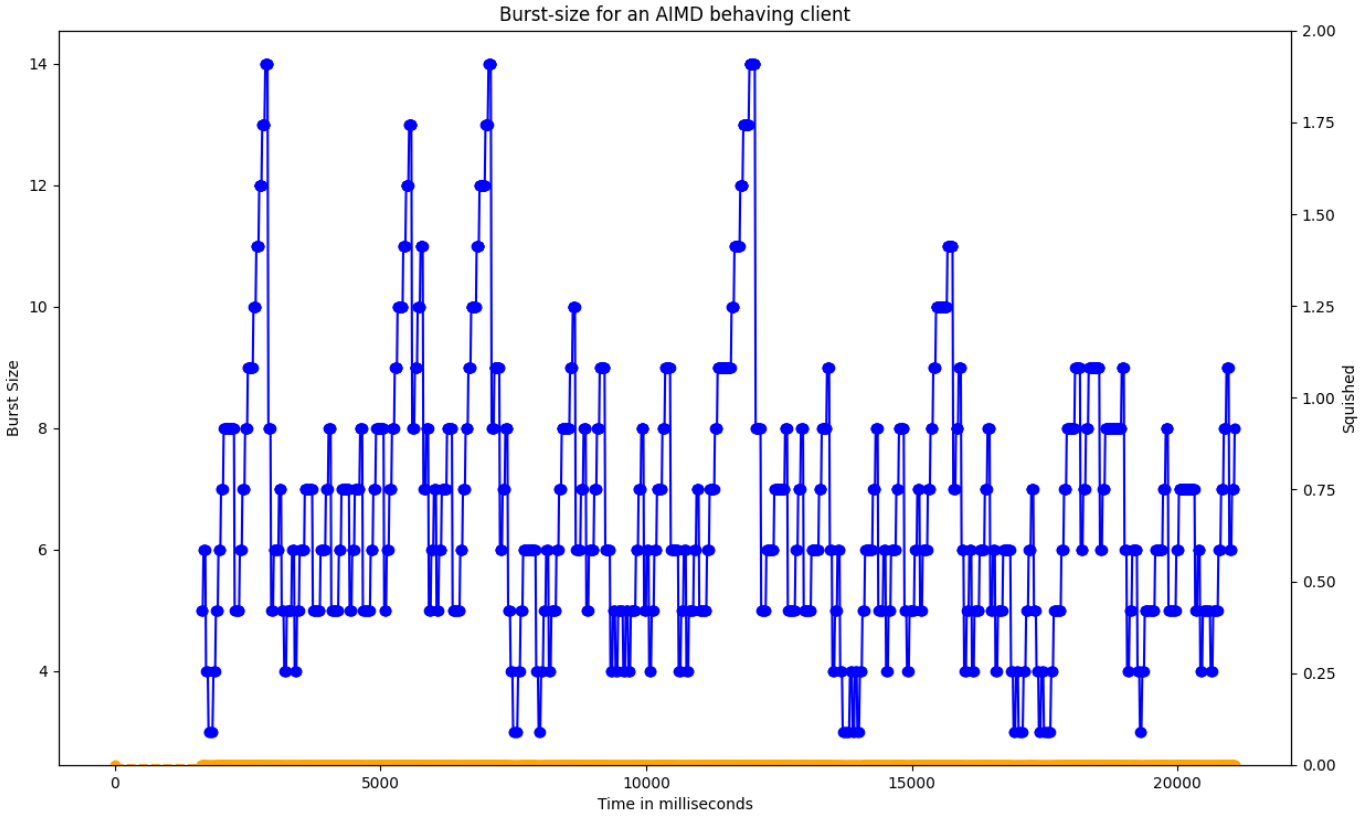

Figure 4: Burst-size in a constant rate server

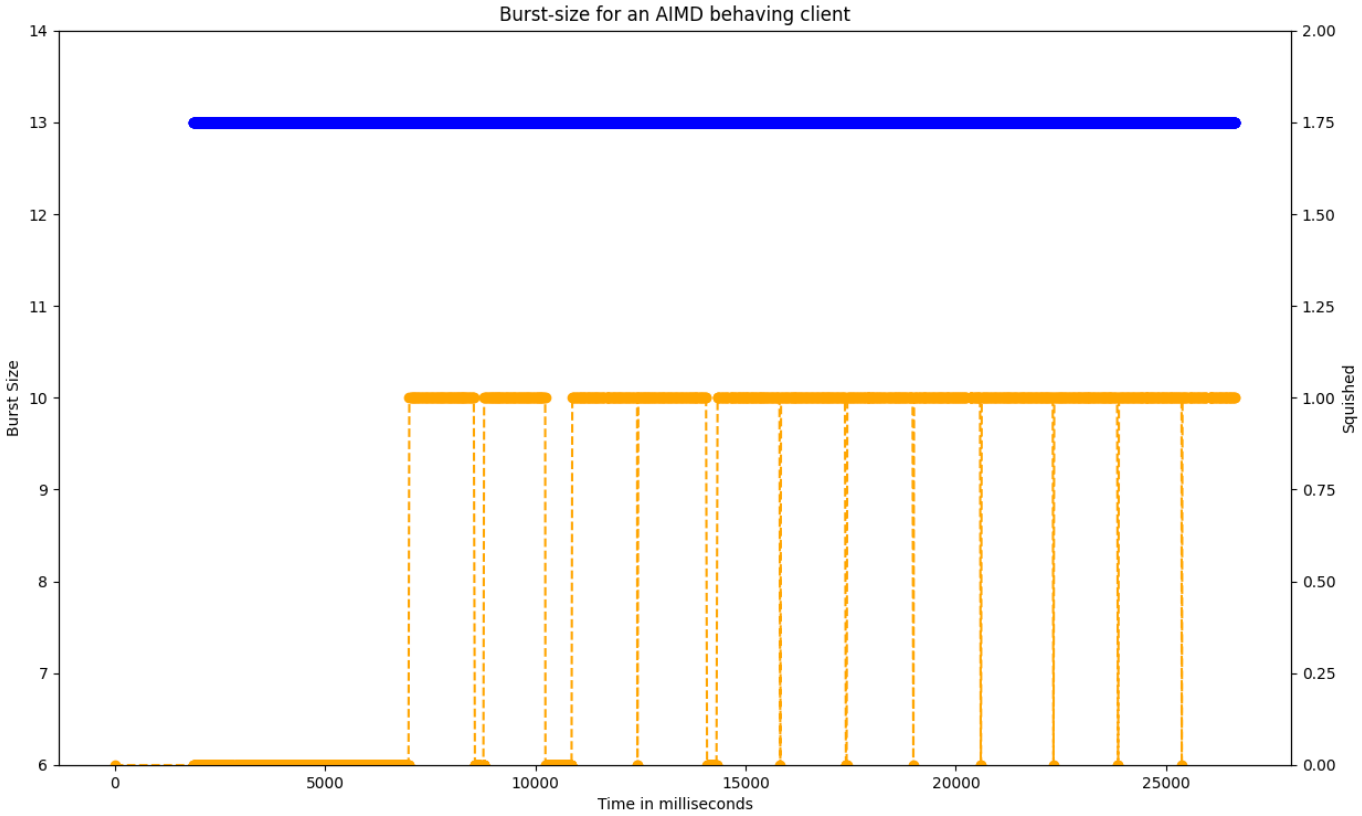

Figure 5: Burst-size for a constant Burst-size client

3 Graph Analysis

Time Taken	Penalty Incurred
20.329 sec	8
18.894 sec	7
24.012 sec	1
21.359 sec	0

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

Time Taken	Penalty Incurred
19.014 sec	13
15.565 sec	41
18.459 sec	20
17.591 sec	15

Table 2: The following data is for localhost server

- 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. Intially we have sent a burst of 5 requests and all 5 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 maintained this burst and continue getting data.
- In third graph we can observe less than 80 percent of burst has come when we requested for 8 packets but only 6 packets have been received so burst size reduced to (8+3)/2=5.
- From Figure 4, it is evident that the system effectively prevents server congestion by maintaining an optimal rate of request transmission. This rate strikes a balance between being neither excessively slow nor overly fast. The ability to regulate this rate is crucial for ensuring that the client does not overwhelm the server, causing it to become "squished."
- We can infer from the graph that most of time burst-size is around 8 i.e whenever we are requesting 8 packets we almost received all the 8 packets. Burst-size graph show the classic AIMD saw tooth behaviour.
- Figure 5 illustrates the performance of a client utilizing a constant burst size of 13. This client experiences a "squish" event at approximately 8000 milliseconds. Even though the server imposes a squish penalty on this client after only about 100 requests, the client's failure to reduce its request rate during the squish period leads to further penalties. Consequently, it faces another squish event due to its failure to slow down, resulting in a subsequent penalty. 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 blocking.