COL334: Assignment 3 Part II

Ashish Arora, Kashish Goel

October 2023

1 Objective of the Assignment

The aim of this assignment was to replicate a TCP-like protocol implementation using UDP connection. The client aims to fetch data from a constant rate server, in a lossy network, and recompile the data at its own end. The server given in this part was a constant rate server, i.e., it sends data at a constant rate. However, if we send requests for data faster than the server rate, we are awarded a suitable penalty for the same.

2 Our Approach for the client

Partitions: We are calling a packet consisting of data from offset to offset+packetCapacity as a partition. The total number of partitions is determined using the total size of the data being received and the maximum size of packet of data that can be fetched.
We are using 3 sets- `remainingPartitions`, `inTransit` and `receivedPartitionSet`. Each of these have their elements as partition numbers.
A partition number present in `remainingPartitions` indicates that the partition is left to be received and that the request for receiving the packet having that partition number is also yet to be sent.
A partition number present in `inTransit` indicates that the request for the particular partition number has been sent to the server but the packet hasn't been received yet.
A partition number present in `receivedPartitionSet` indicates that the packet for particular partition number has been received.
Sending and Receiving requests are handled using threading. First the size of the data is fetched using the function `getTotalSize()` which initiates two threads- one for sending the request to fetch the size of the data using the `sendSizeReq()` function and the other for receiving the packet using the function `recvSizeReq()`.

Approach followed for sending the packets: When the function `sendToServer` is started, it sends the requests corresponding to the offsets, at a rate which is varied accordingly. For every request we send, we append the partition number to the set `inTransit`, indicating that the request for that offset number has been already sent and we are waiting for response from the server corresponding to that offset. We are also fetching the replies from the server for these requests in parallel. Following are the conditions set while sending the requests:

- **Rate and RateSize:** In our implementation, Rate size determines roughly the number of requests that would be sent in a second as rate is defined as the reciprocal of rate size. When the server is squished the rate increases by the squish factor.

- **Maximum Size of the set inTransit:** The set `inTransit` stores the partition numbers for which the requests have been sent but replies are awaited. We are not letting the size of this set exceed the limit of `inTransitSize`.

- **If len(inTransit) > inTransitSize:** If the size of the set is being exceeded, we conclude that the either the packet has been dropped by the server or we have exhausted the tokens in the bucket.
  To check if this has happened due to server loss or 0 tokens in the bucket, we re-send the requests twice for the partitions in the set `inTransit` again separated at the current rate. If the size of the set still does not reduce we conclude that **we have exhausted tokens in the bucket**.

- **Exhaustion of tokens:** As mentioned above, if we conclude that tokens have been exhausted, we decide to decrease the rate size and hence increase the rate, i.e., the time between sending of 2 requests. If such a case is detected, then we decrease the rateSize and determine it using max(rateSize//1.5, originalUpper * 0.3) where originalUpper is the max initial peak rate. Along with this, we also check for certain conditions and update rateUpperLimit and upperFactor if the conditions are satisfied (discussed in Section 3). We also check for the squish state in which case we ensure that the rate is reduced by a certain squishFactor.

The function rateIncrease(): This function is used to alter Rate size and rate when a new packet is received from the server. The call to this function is hence made in the receiving function, whenever a reply is received. Following is the strategy used to alter the rate in this function:-

1. If `squishState` is true, i.e., we are currently squished we increase the server rate to `squishFactor` times the reciprocal of rate size which has already been reduced by a factor when server first squished us.

2. If the length of set `inTransit` doesn't exceed its maximum size and number of newly received new acknowledgements is equal to the transit window size, we aim to increase the rate size. This idea of increasing the rate size is similar to the idea of **Congestion Window** where the window size is increased after 1 RTT if N packets of the window are received timely.

3. **Increasing rate in a probabilistic manner:** If the rate size is less than the upper rate limit set times the upper factor then the rate size is increased additively by 1 and rate is accordingly reduced. If however it is more, rate seems to be sufficiently low and since we don't want to overwhelm the server with requests, we increase rate size only with a probability of 5 percent.

Hence, rate is increased keeping in mind the various possible state of the server and the network.
Approach for receiving packets from the server: Packets are continuously received on the thread running the function `recvFromServer()`. When a packet is received from the server, it is broken down into its various components using RegEx. The partition number for the packet is determined using its offset and the size. Correspondingly, the partition number is removed from the set `inTransit` and added to the set `receivedPartitionSet`. The received data is also added to the list `receivedPartitions`, correctly positioned according to its partition number. As consistent with the idea of rate control mentioned above, with every reply we get, we call the function `rateIncrease()` which increases or decreases the rate accordingly
If we have been Squished by the server then we set the Boolean variable `squishState` to True, else it remains False. When squishing for the first time, i.e., for the first request in the 100 requests we are squished for, we reduce the rate size by a factor of 1.8 while ensuring that it doesn't reduce below 100 correspondingly to a rate of 0.01. We get to know the same by the presence of "Squished" in the reply header from the server.
Packets are received from the server until we get all the packets. Once we get all the packets, we exit the function.
Submitting the final data: Once all the packets are received, the entire data is compiled and the MD5 hash for the same is calculated using the inbuilt functions in python. To submit the final hash, the function `submitFinal(hash)` is called which initiates two threads- one for sending the final hash using the function `sendFinalHash(hash)` and the other for receiving the response to the submitted entry.
The final hash is sent if the flag `finalflag` is false, after forming an appropriate packet. After sending the hash, once it is received the flag is made true. The final submission is sent repeatedly to the server at a constant rate of 0.1 seconds, until the server receives it and the flag `finalflag` is turned true.
After submitting the final hash, the socket is closed.

3 Determining the optimal rate and tuning the parameters

To determine the optimal rate we used the following strategies and parameters:-

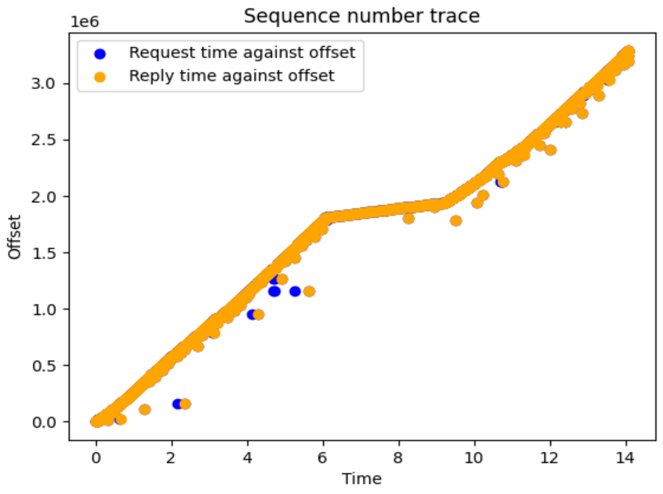
- **Analogy with TCP Self Clocking Idea:** Just like TCP's self clocking idea wherein a new packet is sent in the server with every new acknowledgment, once the set `inTransit` reaches its maximum size, we send a new request only with a new reply.

- **Usage of Rate Size and rate:** Rate size determines roughly the number of requests that would be sent in a second as rate is defined as the reciprocal of rate size.
- A combination of the parameters `inTransitSize` and `rateSize` is used to bring implementation closer to maintaining a **Congestion Window** as we change the rate only on receiving new replies equal to intransitSize and by using rateSize we aim to send a certain number of requests in certain time interval analogous to 1 RTT.
- **Rate Upper Limit and Upper Factor:** We determine the potential upper bound on rate for subsequent requests based on upper limit. Upper limit is set as the peak rate we reach initially. Since the server has full number of tokens at this point, we will obtain the global maxima here. For subsequent requests we set an upper factor starting from 0.7. The product of Rate Upper Limit and Upper Factor gives us a upper bound on the rate we expect the subsequent requests to reach. However if this rate exceeds, then we change the Upper Factor by -0.03 everytime this happens. In case, the rate exceeds and has to be reduced more than 0.45 times Rate Upper Limit, then we update the Rate Upper limit and set an Upper Factor of 0.9.
- **Usage of Probabilities:** Since we are working on a lossy UDP Server, we don't know if the requests have been dropped due to exhaustion of tokens or due to packet drop by server. We hence use probabilities wisely to increase or decrease the rate after experimenting well on the parameters. Apart from this, as discussed above we are estimating upper bounds based on peak initial rates, however since they are "estimates" we don't want to get stuck at a point, but rather allow the rate to increase beyond this bound. For this we apply probability, allowing rateSize to increase with a certain probability above the upper bound estimates while also keeping the penalty in check.
- **Estimating RTT and timeout:** We estimated the RTT and timeout. For every request sent, we stored the sending time and once we received the request we recorded the time reporting the difference as the sample RTT and then using the EWMA formula to estimate RTT where alpha = 0.125. Timeout was calculated using the deviation RTT and estimated RTT. The above parameters were used to fine tune the other parameters we set in our implementation.
- **Usage of constant rate server to determine the approximate rate of the server:** Using the code from milestone 1 where the client was sending requests at a constant rate, we approximated the approximate rates of the server and used that to initialize the rate size and rate.

4 Results and Graphs

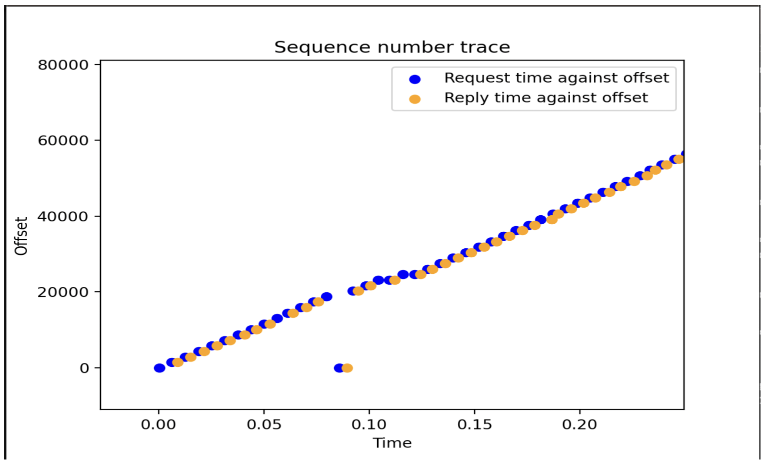
Following were the approximate time and penalty received after running the server multiple times:
Time: 12.5 seconds
Penalty: 7
Following plots were made for our client implementation.

1. **Plot for Sequence Trace:** In this plot, the request offsets and the reply offsets where plotted against time at which they were requested or received.

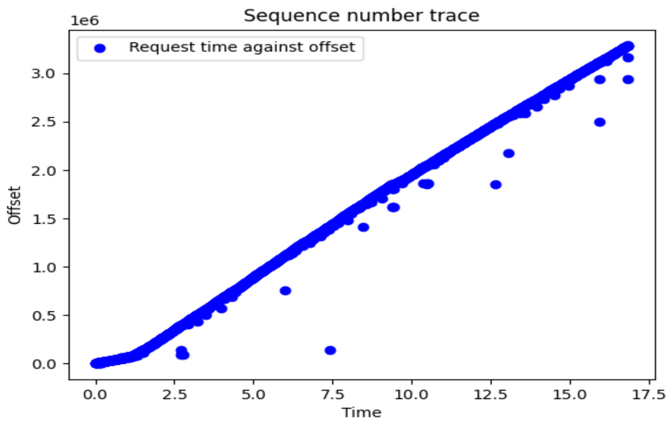


   As shown in the plot, blue points depict the request time vs offset and the orange points depict the reply time vs offset. Replies overlap almost completely with the requests due to the scale of the axis and also the implementation.
   As consistent to the implementation, it can be seen that for most of the offsets for which the request is sent, the replies are received almost instantly. For however, some of the offsets for which the requests would have been dropped by the server, the requests were sent again until the packet was received.

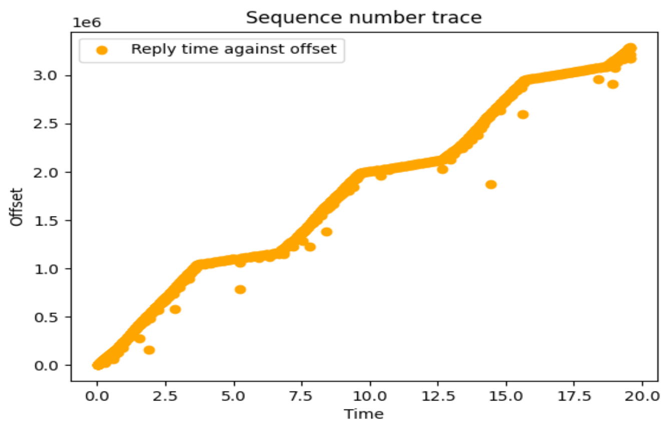
2. **Zoomed-in version of the above graph:** In this plot, the zoomed in version for initial 0.2 seconds has been displayed.



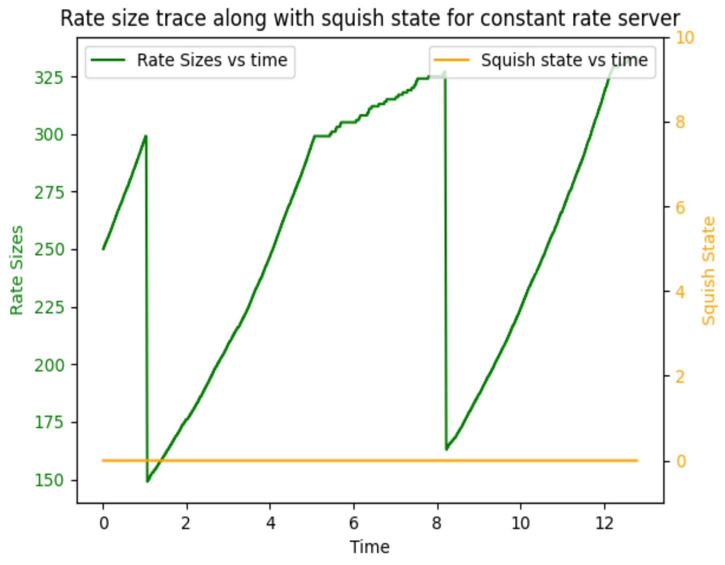
3. **Request offset against time plot:** The plot for the offsets requested against the time they were requested is as below:



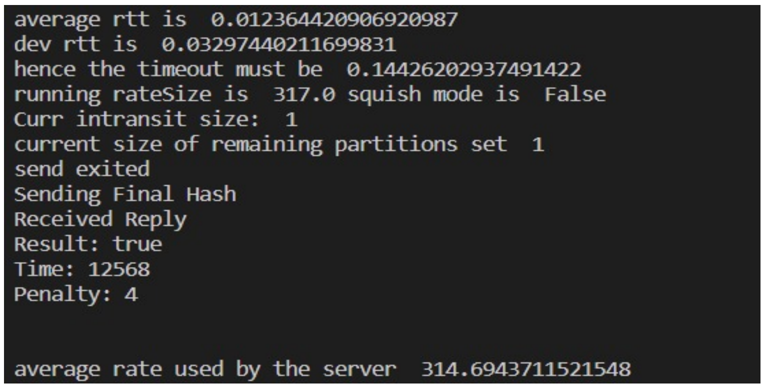
4. **Reply offset against time plot:** The plot for the offsets received against the time they were received is as below:



5. **Plot for Rate size vs time plotted against squish time:** The plot below depicts the rate size used against time with the squish time too plotted.

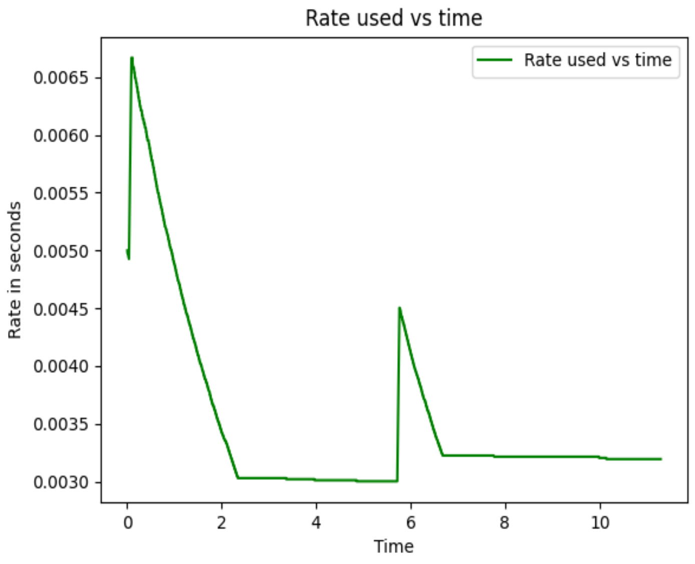


The result obtained from the server for this particular run of the client is shown below:



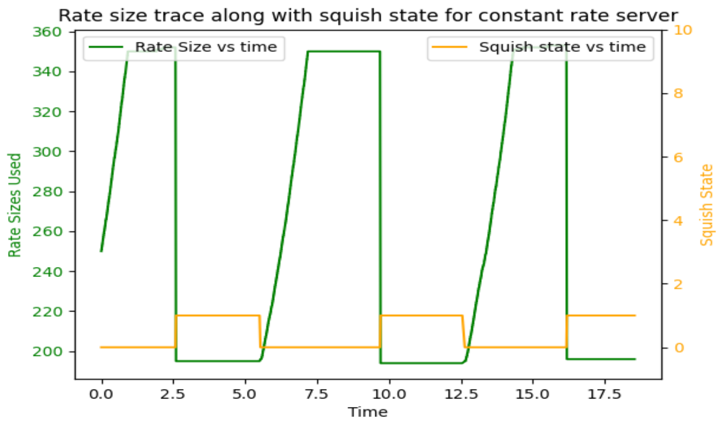
```
average rtt is  0.012364420906920987
dev rtt is  0.03297440211699831
hence the timeout must be  0.14426202937491422
running rateSize is  317.0 squish mode is  False
Curr intransit size:  1
current size of remaining partitions set  1
send exited
Sending Final Hash
Received Reply
Result: true
Time: 12568
Penalty: 4


average rate used by the server  314.6943711521548
```

6. **Plot for Rate vs time plotted:** The plot below depicts the rate used against time.



As visible in the plot, the rate is reduced when the server receives the replies smoothly and the it is increased when it may have seen some reduction in the reply time of server.

7. **How Rate Size is altered when the client is squished:** The below curve shows how rate size varies when client is squished.


Rate size trace along with squish state for constant rate server
Rate Size vs time
Squish state vs time
Rate Sizes Used
Squish State
Time
360
340
320
300
280
260
240
220
200
10
8
6
4
2
0
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5