COL334: Assignment 3 Part III

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 variable rate server, in a lossy network, and recompile the data at its own end. The server given in this part was a variable rate server, i.e., it sends data at a rate that varies with time. We are awarded penalty if we send the data faster than the server rate and exhaust the tokens given to us.

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.
  It may be the case that we had exhausted all tokens and by the time we sent requests, 1 or 2 new tokens were created and hence we received the replies for the requests we had sent. To ensure that we are sending new requests only if packet drop was due to lossy nature of the network, we ensure that we receive the replies of all the requests sent but atmost 1. Since `inTransitSize` has a small value, it is safe to assume that if the tokens have not been exhausted, in two iterations of re-sending the requests, the size of the set `inTransit` would not be greater than 1.
  If the size of the set is still greater than 1, 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 by taking into account currhighcount (discussed in Section 3). If currhighcount is greater than equal to 1 then $rateSize = min(rateSize, currhigh) * 0.85$ otherwise rateSize is equal to $rateSize//1.1$

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. We reduce

the rate size only when we receive “Squished” for the first time in the 100 requests as we don’t want to unnecesarily keep reducing the rate.

2. If the length of set `inTransit` doesn’t exceed its maximum 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:** We change our rate (and rateSize) in a probabilistic manner. The probability depends on the value of currhighcount and rateUpperLimit. In general, we increase the rates faster upto 0.9 times rateUpperLimit or 0.93 times currhigh. Beyond this we decrease the speed with which rate is increased by using probability. This is used to control and make the final rate as close to our estimated value of token generation rate, but at the same time allow for it to adapt to the variations that occur.

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 1.5 (this factor has been decided upon by experimentation and some observations). 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:** Rate Upper Limit is used to set a benchmark to what we think is the upper limit to rateSize. It is initially 10,000 and we first change it after the first request reaches it's peak. Then rateUpperLimit is changed only if currhighcount is greater than or equal to 1.

- **Currhigh and currhighcount**: Currhigh is used to keep track of the last peak that was reached by the rateSize. If the peak reached in the subsequent is within [0.9*currhigh, 1.1*currhigh] then we increment the count of currhighcount by 1, otherwise we set the currhigh to the current rateSize and set currhighcount to 0. The combination of these parameters is used to access and form a buffer region to where we expect the current server token generation rate to be. Since multiple peaks are within the same range, and then fall this means that the optimal rate must also lie within this region. By having these approximate bounds, it helps us in estimating fast we should increase rateSize. If it is much below this range then we increase it quickly however when it is in this range or above it, then we increase it slowly.

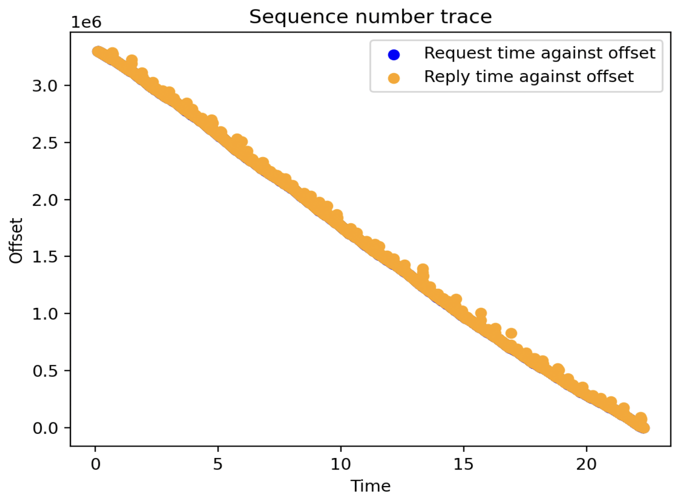
- **Usage of Probabilities:** Since we are working on a lossy UDP Server, we don't know if the requests have been dropped due to exhaustion of tokens or due to packet drop by server. We hence use probabilities wisely to increase or decrease the rate after experimenting well on the parameters. Apart from 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.

4 Results and Graphs

Following were the approximate time and penalty received after running the server multiple times:
Time: around 22 seconds
Penalty: on average around 30
Following plots were made for our client implementation.

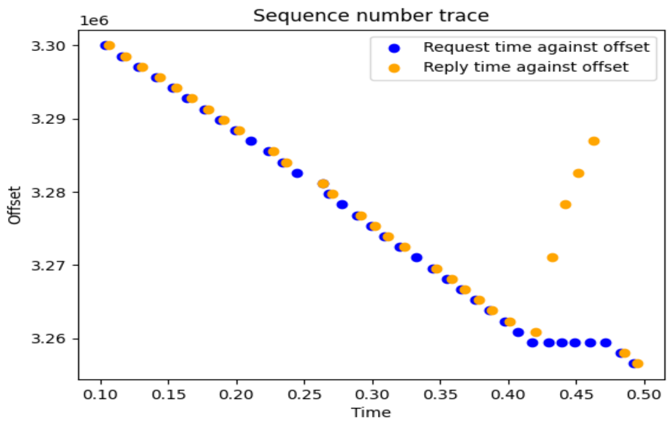
1. **Plot for Sequence Trace:** In this plot, the request offsets and the reply offsets where plotted against time at which they were requested or received.



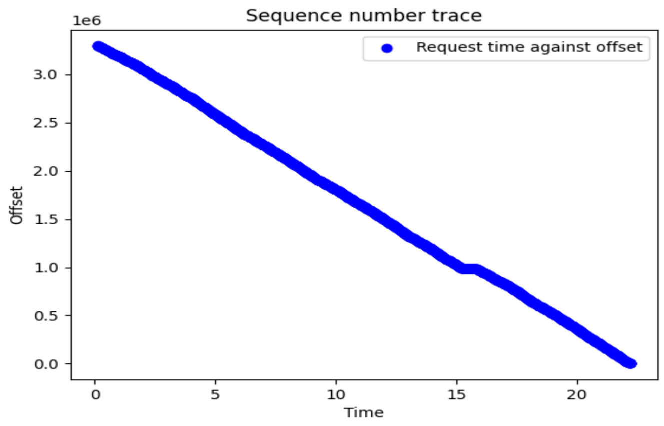
As shown in the plot, blue points depict the request time vs offset and the orange points depict the reply time vs offset. Replies overlap almost completely with the requests due to the scale of the axis and also the implementation.
As consistent to the implementation, it can be seen that for most of the off-

sets for which the request is sent, the replies are received almost instantly. There are rarely any offsets for which replies are not received initially as we ensure that by putting the condition of size of set `inTransit` to be atmost 1.
Note: We are requesting from the highest partition number, i.e., we are requesting the higher offsets first and hence in the curve with time, the offset is reducing.

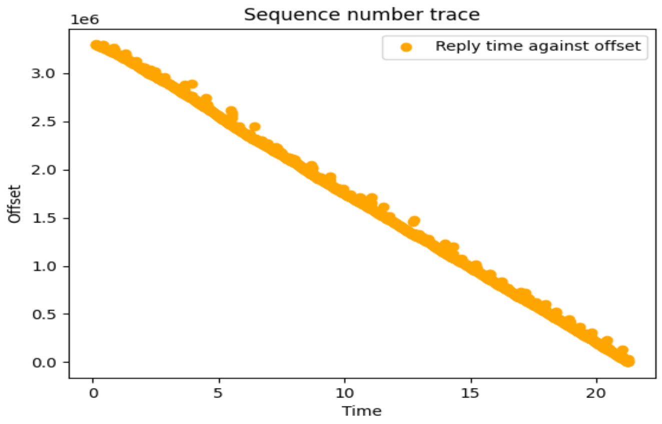
2. **Zoomed-in version of the above graph:** In this plot, the zoomed in version for initial 0.5 seconds has been displayed.



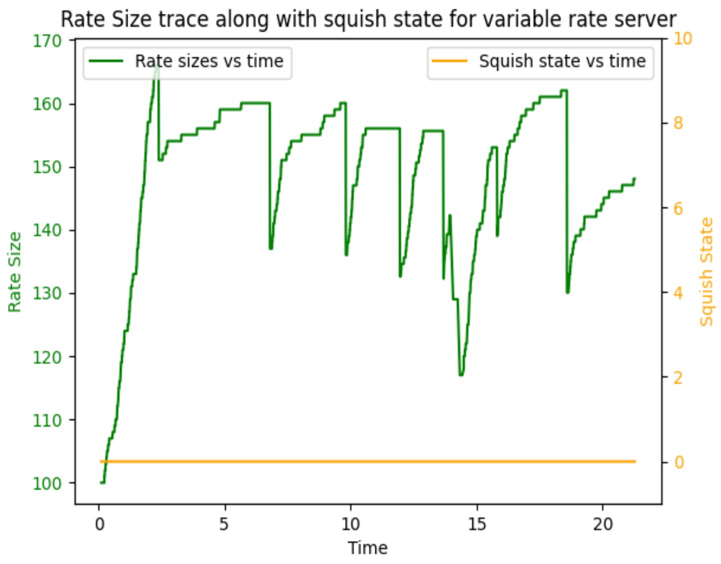
3. **Request offset against time plot:** The plot for the offsets requested against the time they were requested is as below:



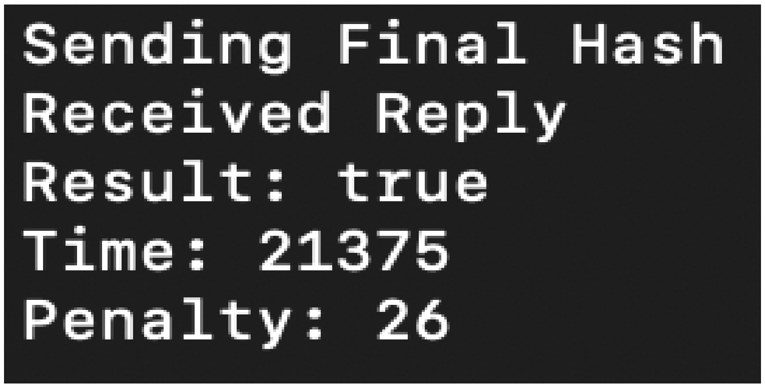
4. **Reply offset against time plot:** The plot for the offsets received against the time they were received is as below:



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.



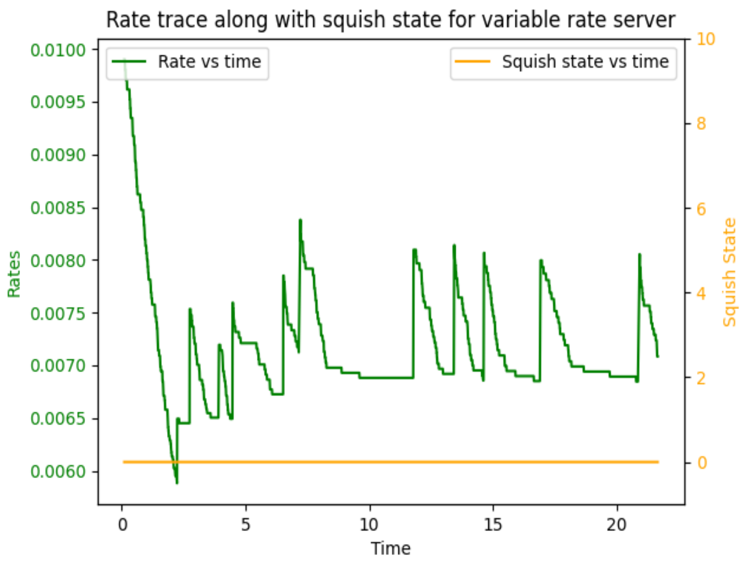
The result obtained from the server for this particular run of the client is shown below:



```
Sending Final Hash
Received Reply
Result: true
Time: 21375
Penalty: 26
```

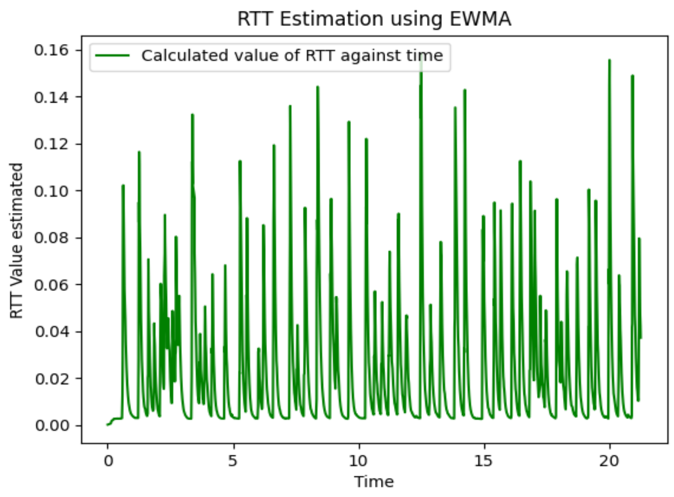
Explanation of the curve obtained: As consistent with our implementation which is similar to AIMD Approach, the graph has a **saw-tooth** kind of shape. `rateSize` is varied according to the situation of the network estimated.
There is a rapid increase in the Rate size initially as the token bucket is full and all the replies would have been received hence leading to increassed number of calls of the function `rateIncrease()`. As discussed above, once we get an idea of the upper bound of the rate size, we increase the rate size slowly around it and hence very slow increase can be seen around the peaks.

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



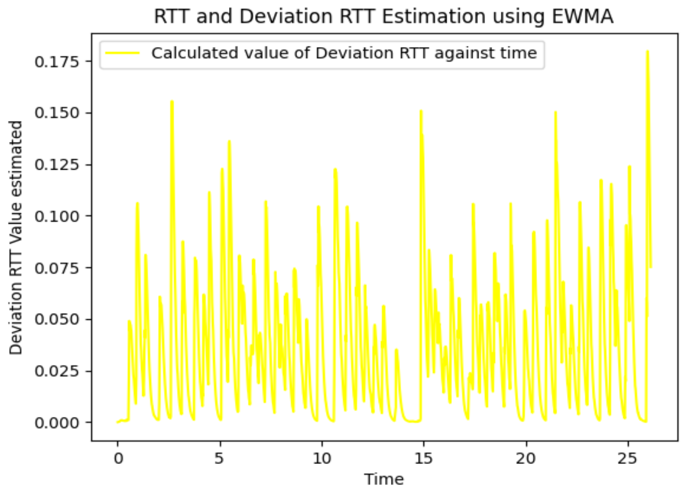
As visible in the plot, the rate is reduced when the server receives the replies smoothly and the it is increased when it may have seen some reduction in the reply time of server. Contrary to the rateSize vs time curve, rate vs time curve has an **inverted saw-tooth** behaviour.

7. **Plot for Estimation of RTT vs time:** We estimated the round trip time using the **EWMA Approach** wherein we kept track of the time taken to receive the reply of every request sent and the value of alpha was kept as 0.125.

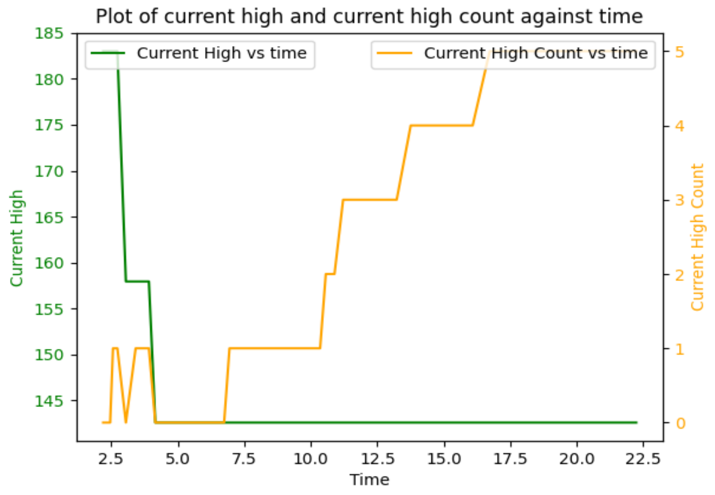
   The plot above is estimating RTT for vayu server with the ip address "10.17.51.115".
   Average value of RTT calculated from the above plotted points was **0.02451 secs**.

8. **Plot for Estimation of Deviation RTT vs time:** We estimated the round trip time using the **EWMA Approach** wherein we kept track of the time taken to receive the reply of every request sent. We also calculated the deviation RTT using the same approach where beta was kept as 0.125.



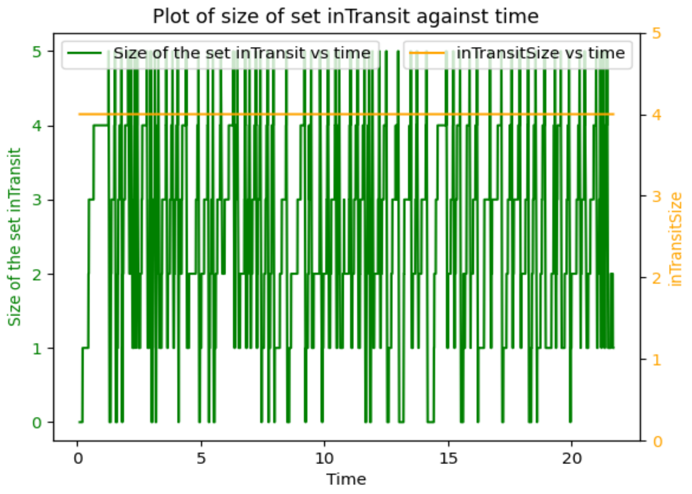
The plot above is estimating RTT for vayu server with the ip address “10.17.51.115”.

9. **Plot for Currhigh and currhighcount against time:** The plot for the two variables explained above are as follows:



As we can observe, over time current high settles to a value and current high count value increases which tells that the subsequent peaks are within our estimated range of [0.9*currhigh, 1.1*currhigh] and so the token rate generation of the server also lies near this range

10. **Plot of size of set inTransit against time:** The size of set inTransit against time is as below:


Plot of size of set inTransit against time
Size of the set inTransit vs time
inTransitSize vs time
Size of the set inTransit
inTransitSize
Time
0
5
10
15
20
1
2
3
4