COL334 Assignment 3

Pravar Kataria (2021CS10075)
Pushpraj (2021CS50596)

October 25, 2023

Contents

§1 Part 1

§1.1 Introduction

Firstly I will describe the problem statement. Then I will go over my implementation (Broad overview along with some fine details). Then I will add some graphs to explain what I did a bit more visually.

§1.2 Overview of the Problem statement

The problem is as such : We have to make a reliable communication procedure (data receiving and sending) on a medium that is not necessarily reliable.
There is a server that emulates this lossy data transfer. One can connect with the server using UDP connections. The server is unreliable in the sense that it can sometimes not respond to request or even if it responds it might take variable amount of time hence the order of packets received back might not be the same as the order of packet requests sent.

§1.3 Idea and structure of the code

First a SendSize command is sent that asks for the number of bytes that the string contains.
This is separate from the main code, until a reply is received as to how many bytes are there in the string, this request is sent again and again, at an interval of 0.1 seconds (this is an arbitrary choice).
Then we know that at a given instant we can query for atmost 1448 bytes, hence the minimum number of queries/requests needed are $\lceil$(length of string)/1448$\rceil$.
Now we calculate this number, say this is N.
Then we keep adding all the answered queries in a set (queries which are received back in the message) . A query is a tuple of the form (offset, length).
After all queries are answered (the size of set becomes N), we concat the string obtained and submit it.
While submitting also it is taken care that sometimes, the server might drop the submit request or not respond. Therefore until a valid response is not received, we keep submitting the MD5 hashed string.

§1.4 Implementation

The implementation can basically be divided into 2 parts :

1. The Data structures used
2. The time delay value and technique used.

§1.4.1 Data Structures used

One data structure that is used is : set, a set is used which stores all queries that are answered.
At every iteration, the offset to be asked is iterated in a linear fashion, if the query is already answered (checked by the set) then we directly jump on next request the next

request, note that the query is not yet added to the set.
The element in the set is added only after a string is received corresponding to the query.
The other data structure used is array ,this simply stores the 'cleaned-up' result of a query and finally this list is concatenated to produce the unhashed result.

§1.4.2 Timeout used

The first implementation was to wait for a reply for some fixed t seconds and then scrap and move on to the next query.
Note that it might be the case that the reply received and the query asked are different.
Then the code got faster as we decreased the value of t from 0.5 to 0.1 to 0.02 but further reduction in t proved to increase the penalty a lot hence proved to be worse for the overall time.
Then a slightly non-trivial idea of exponential backoff was used, this was just that if a query is unanswered then the timeout limit was increased to twice the original backoff limit, and as soon as it gets received it was returned to the base limit.
Both the graphs are being attached (with and without exponential backoff)

§1.5 Graphs

§1.5.1 Graph of offset vs time line graph

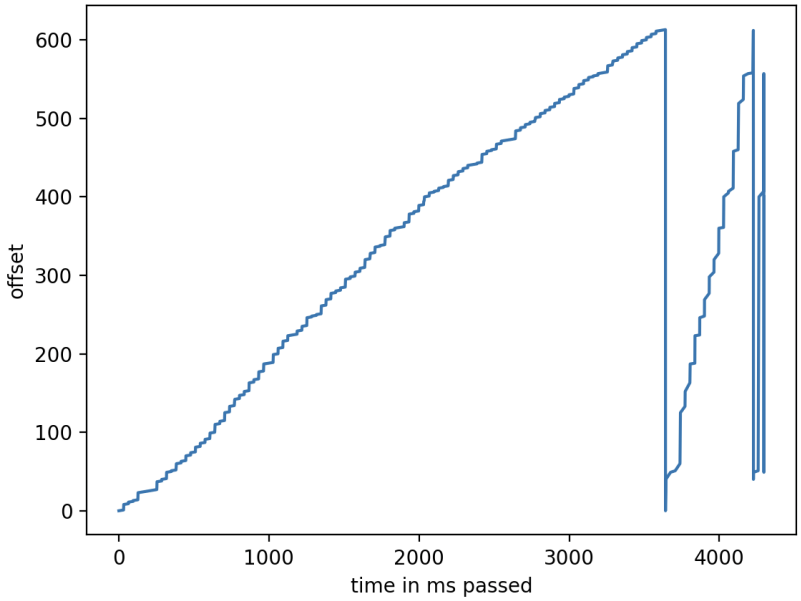


Figure 1: Offset vs time without backoff

§1.5.2 Graph of offset requested vs time (for 0.05s)

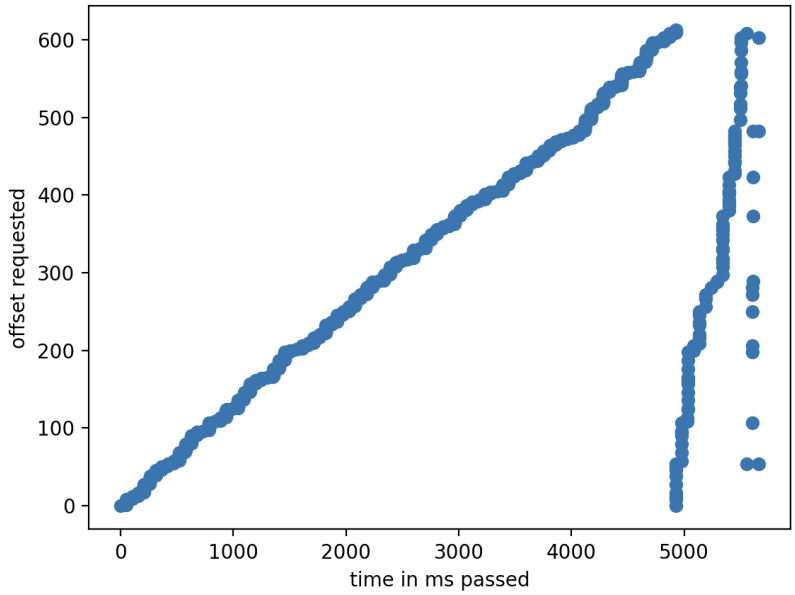


Figure 2: Offset vs time without backoff

§1.5.3 Graph of offset received vs time elapsed (0.05s)

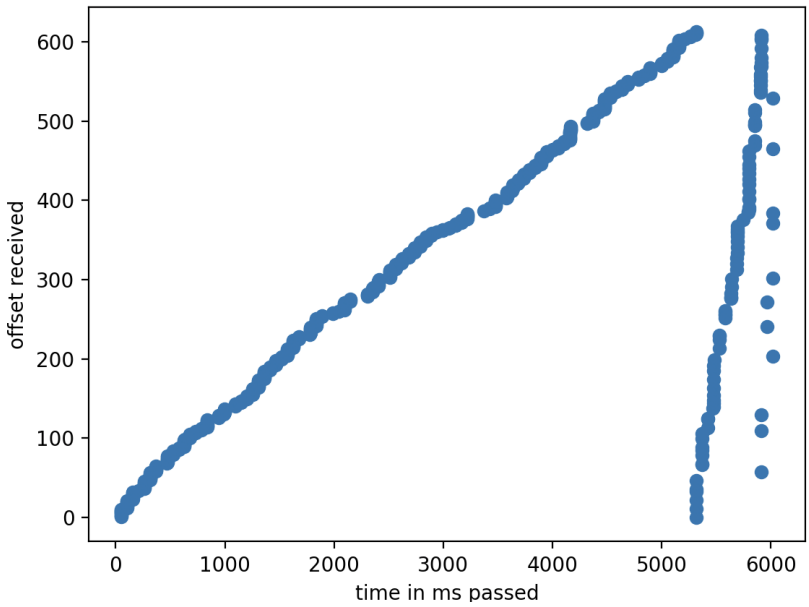


Figure 3: Offset vs time without backoff

§1.5.4 Graph of offset vs time (using backoff method

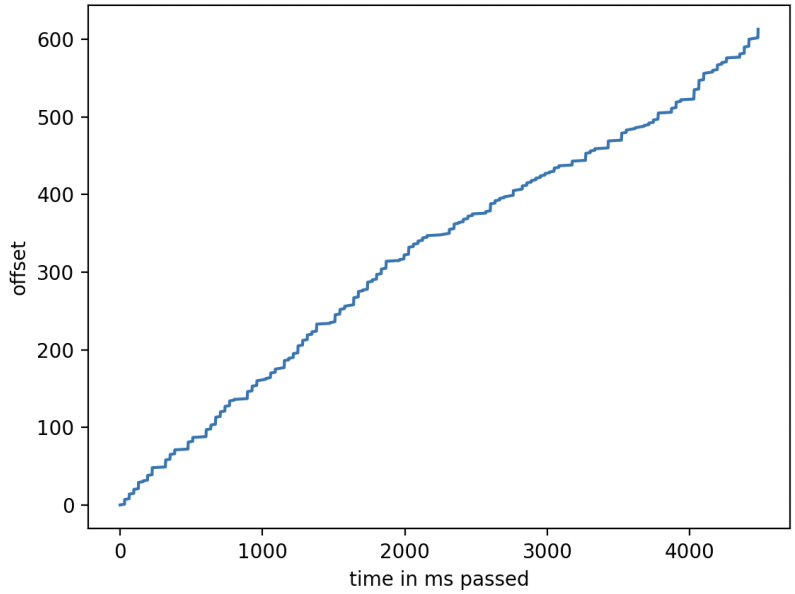


Figure 4: Offset vs time without backoff

§1.5.5 Graph of penalty vs timeout limit

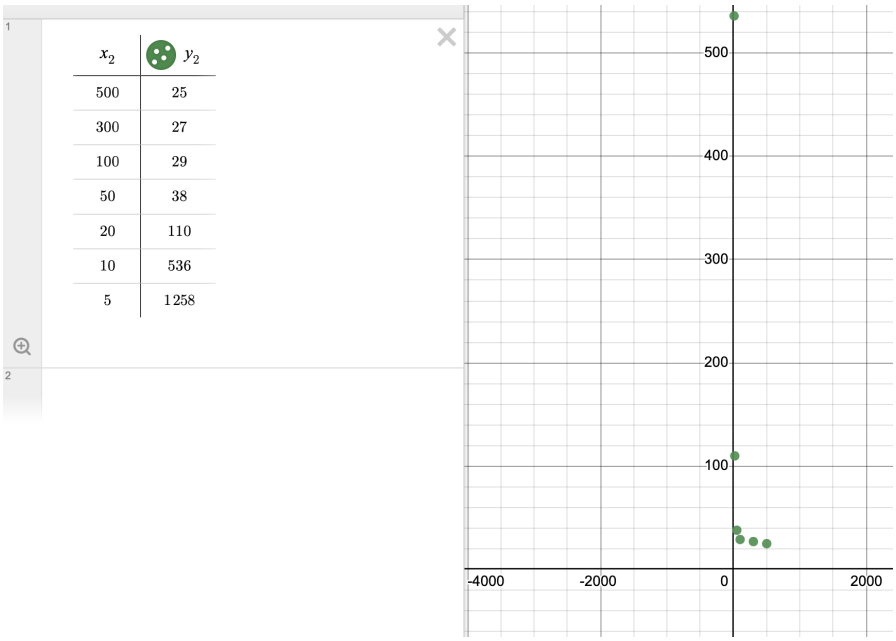


Figure 5: Penalty incurred vs time w

§1.5.6 Graph of total time vs timeout limit

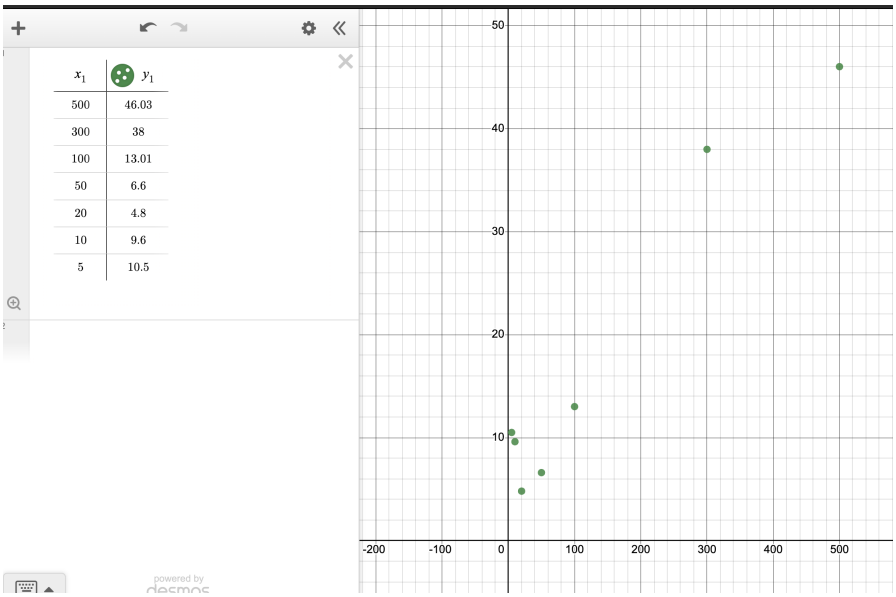


Figure 6: Total time vs time used for backoff in ms

§1.6 Clarifaction of the graphs

For the last 2 graphs, I will attach the tables that are used to make it.

Limit of timeout	Penalty incurred
500	25
300	27
100	29
50	38
20	110
10	536
5	1258

Limit of timeout	Total time
500	46.03
300	38
100	13.01
50	6.6
20	4.8
10	9.6
5	10.5

Also with backoff, we are repeatdly quering the same offset with increasing timeout limits, hence the graph is one perfect line, whereas in the non exponential backoff case (constant time limits) we just keep quering the next.
Also the x axis are in the units of milliseconds.
And also the offset is in units of index of query and not the actual offset value. It is essentially (offset/1448).

By offset requested, I mean the offset that is queried by the server.
By offset received, I mean the offset written in the message received from server.
Note: In this case as the penalty is not too high and the rate is not too 'nonuniform'. The graphs of offset received and offset requested is about the same.

§1.7 Observations

1. Decreasing timeout time increases penalty

2. A sudden drop in the slope of line is of offset vs time is probably due to increasing penalty.

3. Gaps (discontinuities) between the scatterplots is probably due to leaky bucket getting filled and no more requests being replied to.

4. When we increase timeout limit, less iterations of the queries are needed, by which I mean, less iterations of going through from 1 to N are needed.

§1.8 Result

The connection is now reliable and the time taken for 10000 characters (bytes) is 4 seconds (timeout limit used is 0.03 s or 30 ms)

§2 Part 2

§2.1 Introduction

The earlier attempts were sequential, basically no burst sizes, no sleep times, just keep asking asking one request after another.
This has some pros and some cons.
An obvious pro is that it is easy to code and works well for a general setup.
One con is that it does not utilize the information that a leaky bucket is the underlying structure that is maintaining the traffic flow.
One good way to utilize the information is using burst requests, or the classic AIMD method for TCP requests.

§2.2 Overview of the design

§2.2.1 Implementation

The design of the algorithm is as follows :

1. Send a burst of requests consisting of some arbitrary amount of requests.
2. If all the packets are received correctly, then increase the burst size by some amount (increment rate generally 1 or 2)
3. If some packet is dropped in the burst request, then just reduce the burst sending rate by a factor of 2. (reduce it to half the original burst size)
4. Between 2 burst requests, give some sleep of about

$$2 \cdot \text{burst size} \cdot \text{rtt time}$$

(to let all requests be received)

§2.2.2 NOTE :

I was not planning on doing it binary in nature, i.e. reduce the rate to half as soon as 1 packet is dropped.
This is because some packets (about 10%) are dropped even without penalty, just due to uniform packet loss. But in this case where the average burst size is already too low around 6-8, $> 10\%$ penalty is simply equivalent to non zero penalty.

§2.2.3 Some specific implementation details

- The timeout time is set to $4 \cdot$ avg rtt.
- The sleep time between burst is some constant of propotionality times the rtt time
- In one implemenation, this constant of propotionality is exponentially backed off, to say that everytime a packet is dropped amongst the burst requests, along with the burst size being reduced, this sleep time is also increased by a factor of 2.

§2.3 Analysis

§2.3.1 RTT times

Statistic	Value
Count	609.000000
Mean	0.000497
Standard Deviation	0.000765
Minimum	0.000145
25th Percentile	0.000320
50th Percentile (Median)	0.000417
75th Percentile	0.000524
Maximum	0.013565

This is the table of RTT times across the whole transaction with the server (**hosted on local client**)
This is just for making sense of how much the RTT is while transaction on a local server.

§2.3.2 Squishing count vs Timeout time

The timeout time is basically the amount of time waited before a receive request is declared lost.
For example if the time is 0.01 s, the program waits 0.01 seconds on a receive requests before it proceeds to ask for another request.
Increasing this timeout time also reduces the penalty or equivalently the squish count. This is because the timeout is basically a sleep if a message is not received. The larger the sleep period the less the penalty.

§2.3.3 Penalty vs Timeout time

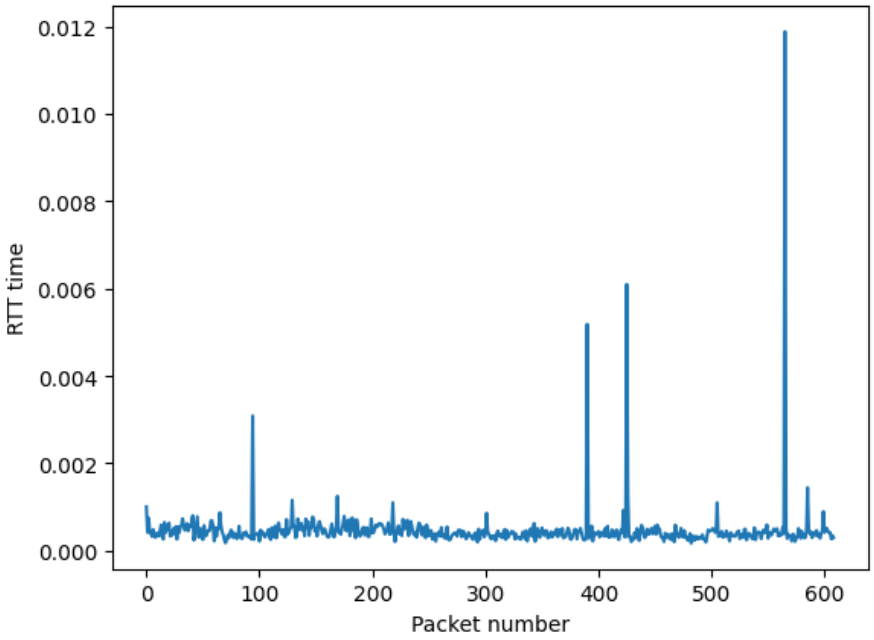
As penalty and squishing count are basically correlated, increasing timeout time decreases both squishing time and penalty.

§2.3.4 Penalty vs drop probability

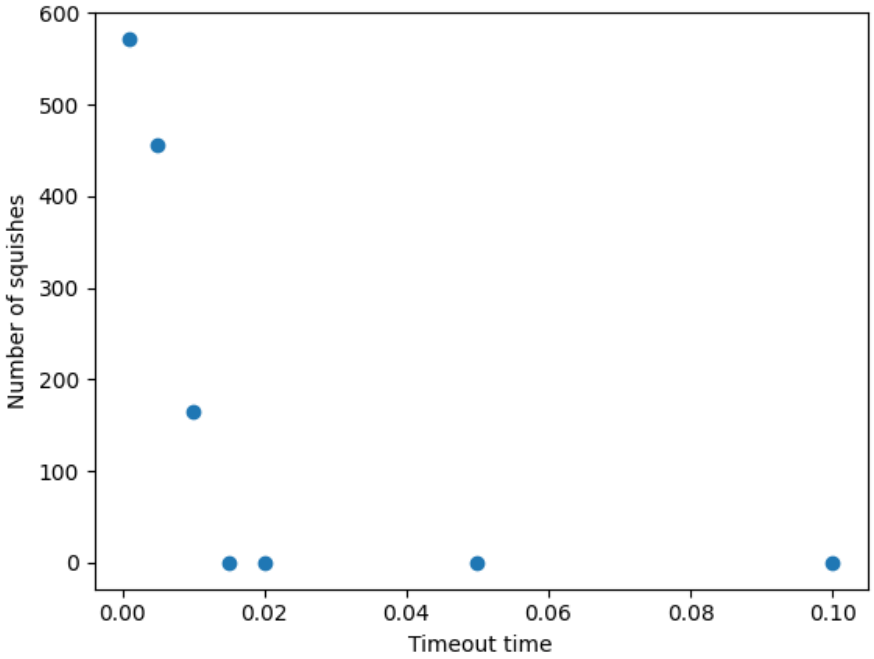
The probability of dropping of request is monotonic with the penalty , modulo the noise. This is because higher the penalty, more number of packets shall be dropped and therefore the probability of drop increases.
But even if the Penalty is 0, the probability of drop is still $\sim$10%

§2.4 Graphs

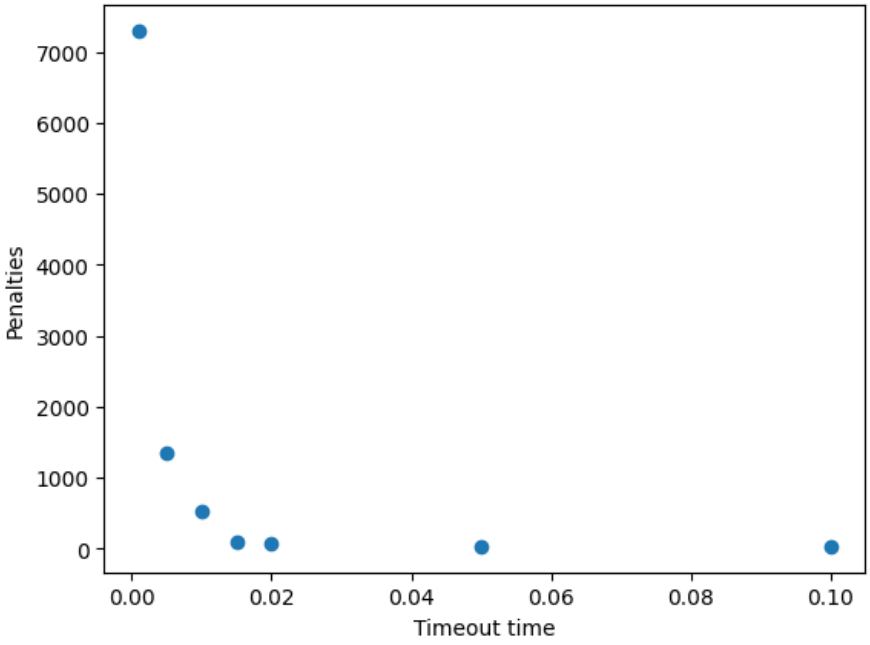
§2.4.1 RTT times



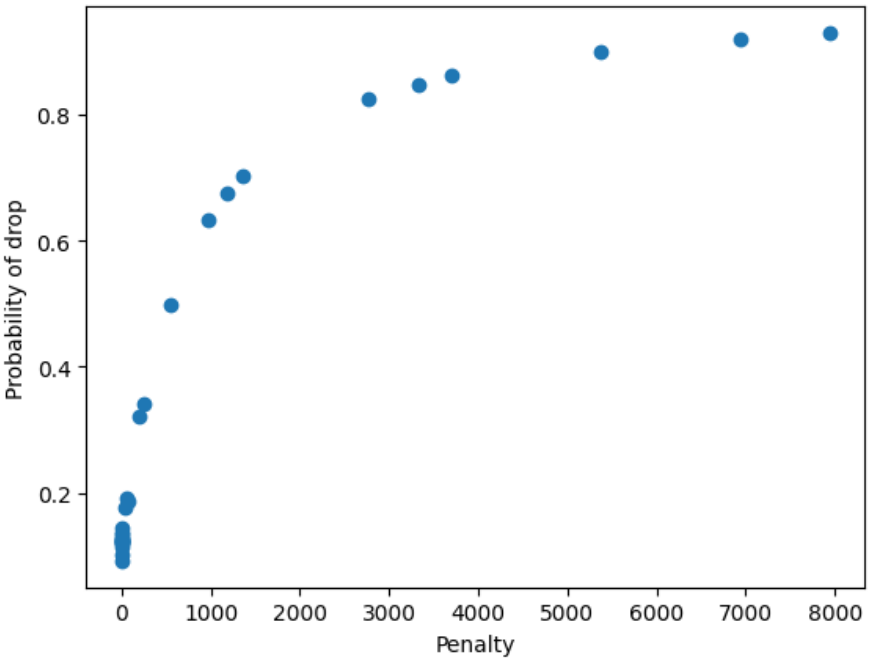
§2.4.2 Squishing count vs Timeout time



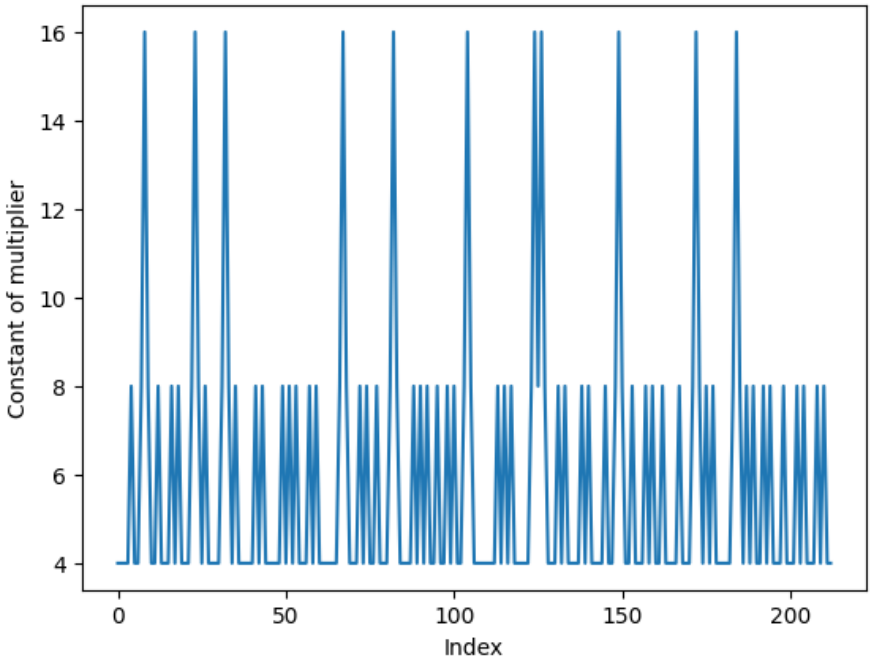
§2.4.3 Penalty vs Timeout time



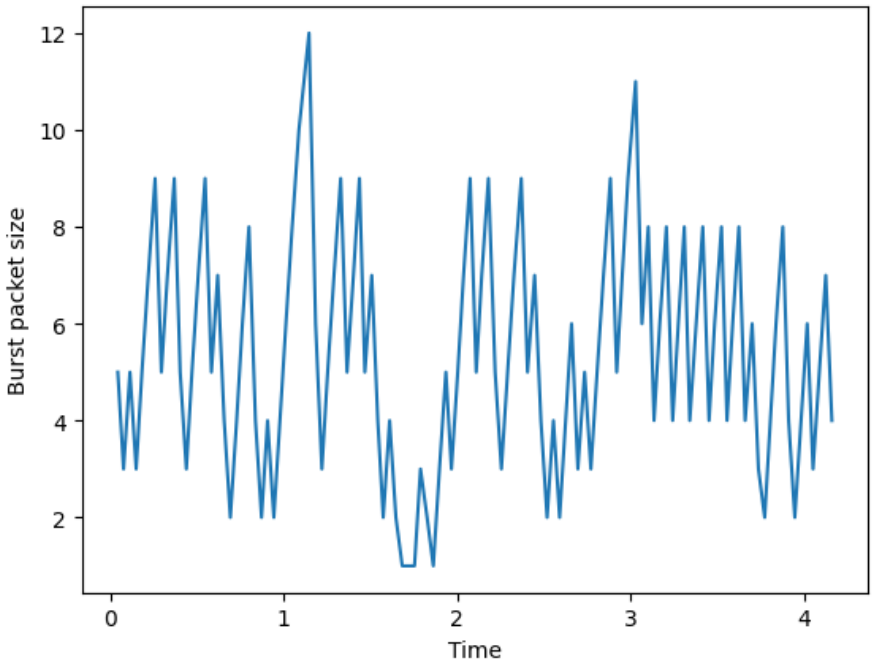
§2.4.4 Penalty vs drop probability



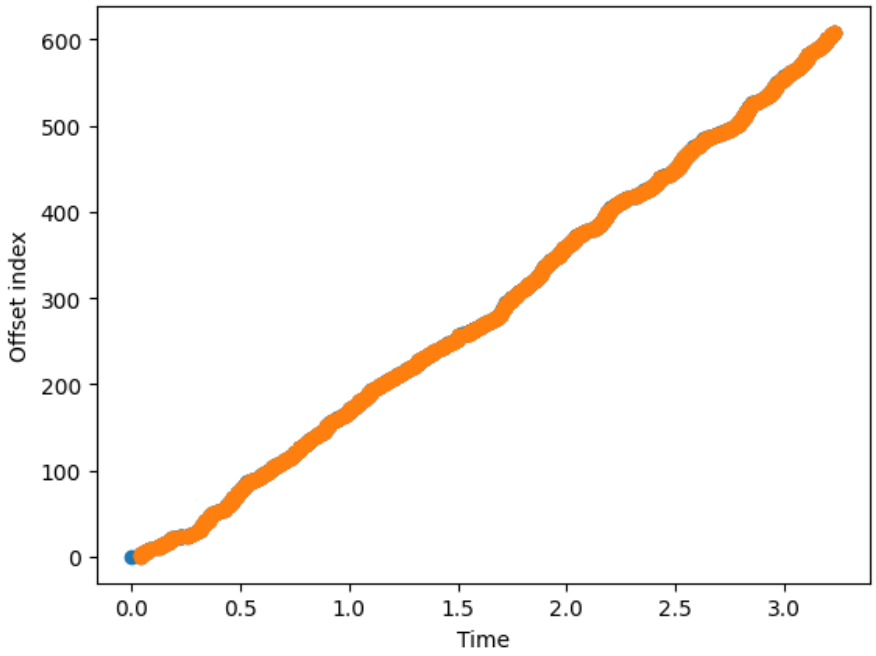
§2.4.5 Varying constant of multiplier



§2.4.6 Burst size against Time



§2.4.7 Time vs offset (unenlarged)



§2.4.8 Time vs offset (enlarged)

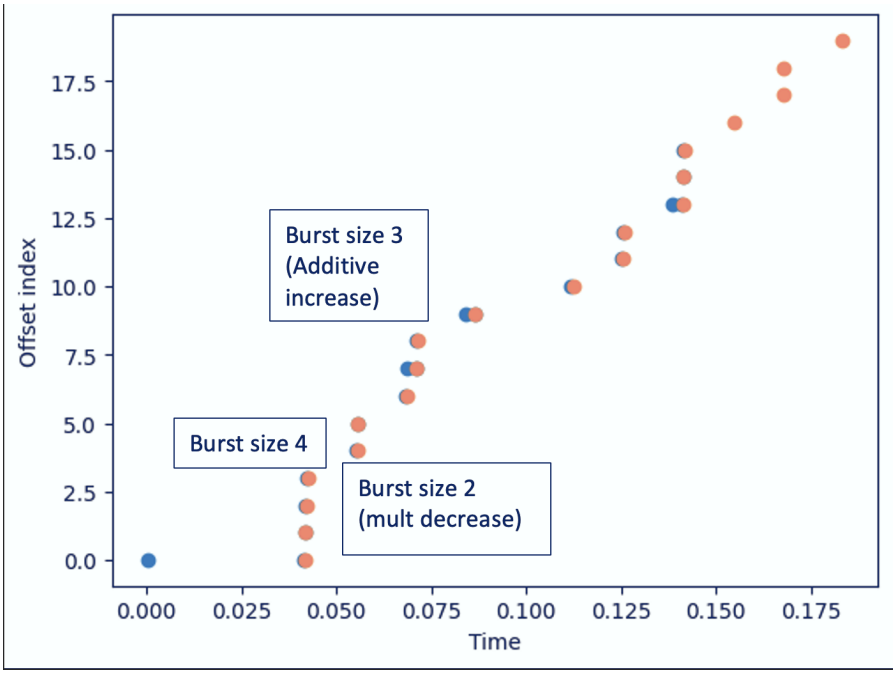


Figure 7: This shows working of AIMD.

§2.5 Explanation of the last few graphs

§2.5.1 Explanation of 2.4.5

The graph is a plot against constant of multiplier vs time. As when exponential backing of in the sleep time is introduced, the constant of proportionality of sleep time vs rtt is changed accordingly, varied exponentially. This graph just shows that, it is increased by a factor of 2, this similar idea might also be implemented for the constant of propotionality for the timeout time relative to the rtt time.

§2.5.2 Explanation of 2.4.6

This graph of Burst size against time is a standard sawtooth graph, the size increases linearly forming lines with slope and then sharply falls (reduction by a factor of 2) and this cycle repeats.

§2.5.3 Explanation of 2.4.7

This is a zoomed out graph of the offset index requested vs time, also on the same graph offset index received vs time is plotted.
To be noted that they almost overlap and cannot be differentialed.
Also for the ease of implementation, the offset is requested until received hence the graph is not like the last time, where the cycle was repeated a few times.

§2.5.4 Explanation of 2.4.8

This graph is just an enlarged version of the previous graph, this shows the exact AIMD in action, the initial burst size is 4, arbitrary, then the burst decreases by a factor of 2 to 2, then in the next iteration increases by 1 to 3.

§2.6 Miscellaneous

§2.6.1 Other things tried

1. Different additive rates, burst size increased by not 1 by but some other constant amount
2. Different multiplicative rates, burst size reduced not by a factor of 2 but by some other amount

§2.6.2 Things that couldn't figure out

1. How to figure out the bucket size accurately (tried by binary search on number of requests before squish)
2. How to find the token replenishing rate before and after penalty/squishing

§2.6.3 Things to be tried

1. Threading
2. incooperate exponential backoff in timeout time limit

§2.7 'Hacky' methods

Some things that were incooperated that may be considered as a hacky fix and maybe not a general method are :

1. Overfitting some things to adjust to the fastest time
2. Explicitly writing some constant values to make the code work correct and fast

Examples of the above points include writing the sleep value as max(some constant time, some times the rtt).
Also the initial burst size estimate must be small otherwise a squish can be encountered in the start itself.

§2.8 Results

The penalty is less than 20 in both remote and local servers for both characters as well as 10000 characters.
The time taken by the code 1 is ∼4s for the 10000 characters and ∼19-20s for 50000 characters.
In the other code it takes 13 seconds with some ∼50 penalty for 50000 characters.

§3 Acknowledgments

1. Socket programming from Youtube
2. Desmos