COL334 Assignment - 3

Milestone-3 Final Report

November 1, 2023

Contents

1 Aim

Milestone-1 of this assignment aimed to ensure that we can transfer data reliably by sending UDP requests slowly. Milestone-2 of this assignment aimed to ensure that we can transfer data reliably while incorporating congestion control mechanisms to the case where the server maintains a constant rate and works under the leaky bucket framework. Because of this assumption, we can optimise our approach to more efficiently receive the needed data on the top of reliability. Our final server, Milestone - 3 has a variable rate server where we need to **construct a generic client that adjusts its burst rate accordingly with the inferred token refilling rate of the server**.

2 Overall Methodology

Adaptive Burst Scheduling: We sent requests in bursts, where the burst size was dynamically adjusted based on the server's responses. We employed adaptive algorithms to quickly adapt to changes in network conditions and server behavior.

Squish Detection: Squished requests were detected by parsing the server's replies. When squished, the client reduced its burst size, preventing further squishes and allowing the server's leaky bucket to recover gradually.

RTT Estimation: We calculated Estimated Round-Trip Time (ERTT) using Exponential Weighted Moving Average (EWMA) to estimate the time taken for requests and replies. This information was used to adjust the timeout duration dynamically, ensuring efficient detection of lost requests.

Data Retrieval: After receiving responses, the client assembled the data from the server into a complete dataset. This was done by organizing the received data fragments based on their offsets and combining them into a single, coherent output.

MD5 Hash Verification: After receiving all data, the client verified the integrity of the received data using MD5 hashing. The computed hash was sent back to the server for verification, ensuring the received data's authenticity and completeness.

Continuous Monitoring: Throughout the process, we monitored various metrics, including burst sizes, request rates, squish periods, and round-trip times. These metrics were used to analyze the client's behavior and network interactions, enabling us to make informed decisions for adaptive adjustments.

3 Our Implementation

We send our requests and receive them synchronously in bursts. This requires two major parameters, a **burst size**, that controls the number of requests made in a *burst* and **time between bursts**. A secondary parameter required is the exception timeout time while receiving data from the server.
The waiting times of our algorithm should be a function of **Token size and R.T.T. (round-trip-time)**. This is because these times ensure that the server does not punish us for making requests too hastily.
From the above implementation and learnings from previous milestones, our client can ensure -:

- Congestion Control
- Reliability
- Speedy file transfer

3.1 Finding out the R.T.T

Initially, we send out about 20 requests and record the rtts for the successful queries. The median among them would be our initialised rtt. Further, it can be adaptively changed by using **EWMA** (Estimated Weighted Moving Average) wherein the rtt is sequentially updated from the old value (biased with a weight $1 - \alpha$) towards the new value (biased with weight α).

$$\text{EstimatedRTT} = (1 - \alpha) \cdot \text{EstimatedRTT} + \alpha \cdot \text{SampleRTT} \tag{1}$$

The waiting time was thus set to be the RTT$\cdot w$ where w is some weight.

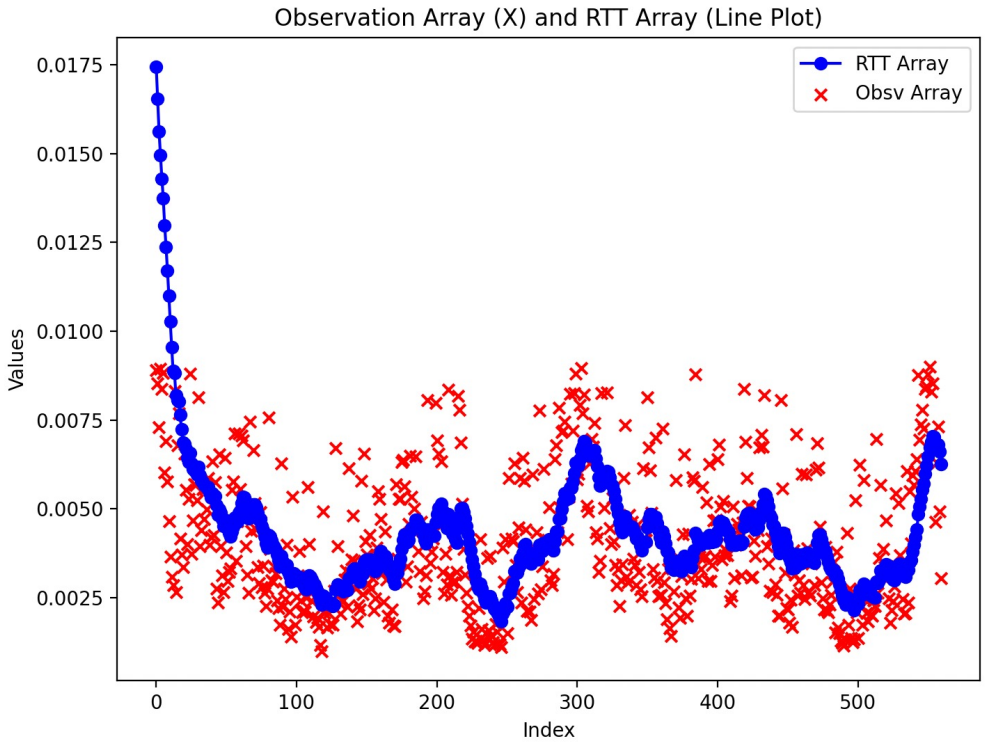


Figure 1: EWMA of RTT along with observed values

This graph shows us how the observed RTT measured on `vayu.iitd.ac.in` helped guide the EWMA RTT towards it.

3.2 Finding the optimal Bucket Size (A.I.M.D.)

Our primary goal includes not getting squished by the server. This would be ensured when the server is being requested values well within the rate of token generation. We keep a track of requests that could not be serviced because enough tokens were not available. The "rude behaviour" metric we take is $\frac{\text{number of un-serviced requests}}{\text{requests sent in a buffer window}}$. We increase our bucket size (additively) whenever we exceed this metric by 0.9 and decrease it (multiplicatively by half) whenever it exceeds 0.9. This ensures that you quickly fall below the allowed rate of requesting if you exceed it, and only try to reach the valid point slowly without exceeding it too much.
Take, for example the current rate of token generation requires a burst size of maximum x and it is decreased to x'. Then the adjustment for bucket size can be achieved in $log2(x - x')$ iterations of bursts.

3.3 Inter-Burst and Timeout Parameters

The time gap between two burst sizes is a function of burst size and RTT. Similar is the case for the time you would wait to try-except the receiving of a packet (sock.timeout()).

`sock.timeout() = burstsize*RTT*0.5`
`inter-burst time:  9RTT/(burst size)`
Explanation - While receiving packets in bursts, the more the number of packets are, faster the tokens get replenished. Thus, a time proportional to RTT and to the burst-size ensures that there is enough time to replenish those tokens during high burst sized queries.
Secondly, for inter-burst times. Given that a burst is over, the number of queries inside the burst do not matter directly. Knowing this, since burst-sizes were previously increased so as to match the replenishing rate of the server, a high burst size implies a higher token refilling rate and thus would require lesser times. Thus inter-burst times are inversely proportional to burst-sizes (and thus have a direct correlation with token refilling rate).

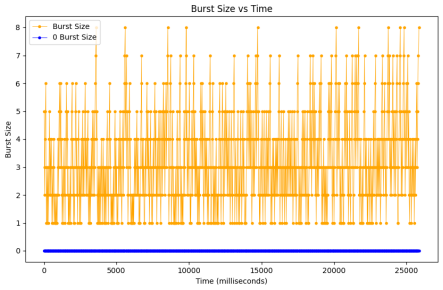
4 Difference between Constant and Variable Rate Server

The difference for a constant rate server is that we could quickly find out the optimal rate of querying. This means that you don't need to keep changing the requesting rate after increasing the burst size up to just below the rate of token regeneration. *In case the rate was varying*, and say it suddenly decreased by x amount, then **multiplicative decrease** would allow you to **quickly fall below that limit in** $\log_2(x)$ **iterations.**
Now for a **constant rate server**, using **Additive Decrease** instead is a smarter option since you do not need to adapt for sudden fluctuations quickly, and you can stay just below the allowed rate in this case.
We never use a multiplicative increase (for any kind of server) since it could far exceed the allowed limit (e.g. if the rate is r, it could jump from $r-1$ to $2r-2$ ie almost double the allowed rate).
To summarise - AIAD would give a faster result for Constant rate server whereas AIMD gives us a better result for variable rate server.

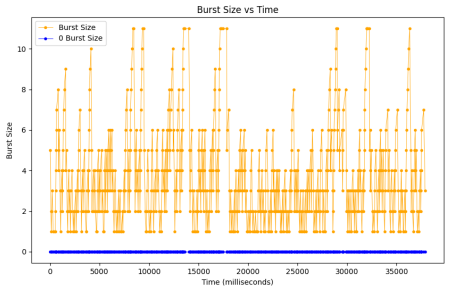
4.1 Clever Hack vs Dynamic Approach when applied on different servers

Table 1: Comparison of Time Taken on a Constant Server

Scenario	**Dynamic Program (AIMD)**	**Constant-Specific Hacky Program**
Time Taken	More as it is not exploitative enough	Less as it exploits the knowledge of constant rate
Squishing Occurrence	Rare, adaptive rate control	rare but penalties are higher
Network Utilization	Efficient, adapts quickly	May overutilize as it is more optimistic but slowly adapts



(a) Finish time = 25s for milestone2 client on constant-rate



(b) Finish time = 35s for Dynamic Client

Figure 2: Exploitative Hacky solution works faster on constant rate server than dynamic client.

Table 2: Comparison of Time Taken on a Variable Server

Scenario	**Dynamic Program (AIMD)**	**Constant-Specific Hacky Program**
Time Taken	Much less as it is adaptive	More
Squishing Occurrence	Rare, 0 on most runs	2-3 squishes
Network Utilization	Efficient, adapts quickly	High overutilisation during congestion

This shows the adaptive model works much better on the variable server.

5 Challenges Faced and Solutions

5.1 Network Latency and Packet Loss

Challenge: Unpredictable network latency and occasional packet loss can affect the reliability of data transfer.
Solution: Implemented adaptive timeout mechanisms and error handling. Used Exponential Weighted Moving Average (EWMA) to estimate Round-Trip Time (RTT) dynamically. Retransmitted lost packets after a timeout to ensure reliable data reception.

5.2 Server Congestion and Squishing

Challenge: The server imposes penalties ("squishing") for rapid requests, reducing token generation rates and causing delays in data reception.
Solution: Implemented exponential backoff for request rate. When squished, the client reduced its request rate, allowing the server's leaky bucket to recover. Adjusted burst sizes dynamically to balance request rates and avoid squishing.
Further, the time gap between two burst sizes is a function of burst size and RTT. Similar is the case for the time you would wait to *try-except* the receiving of a packet (sock.timeout()).

```
sock.timeout() = burstsize*RTT*0.5
inter-burst time:  9RTT/(burst size)
```

5.3 Dynamic Server Behavior

Challenge: The server's leaky bucket parameters and response times varied dynamically, making it challenging to adapt the client's behavior effectively.
Solution: Implemented real-time monitoring of server responses. Adjusted burst sizes, timeout durations, and rate control parameters dynamically based on server responses. Used AIMD approach to adapt to varying server behaviors.

5.4 Optimal Rate Control

Challenge: Determining the optimal request rate and burst size to maximize throughput without overwhelming the server or causing excessive squishing.
Solution: Conducted experiments with different burst sizes and request rates. Used real-time feedback from server responses to dynamically adjust the client's behavior. Analyzed the trade-off between request rate, burst size, and squishing penalties to find optimal parameters.

6 Recordings and observations

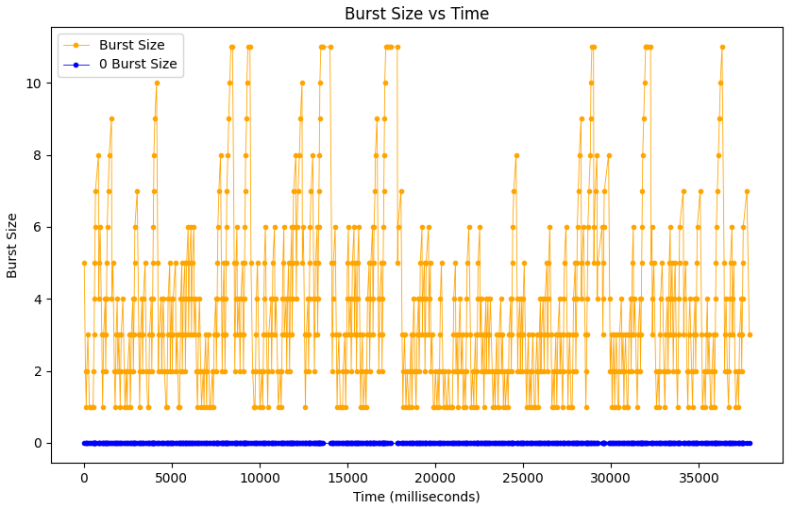


Figure 3: Burst size varying vs time

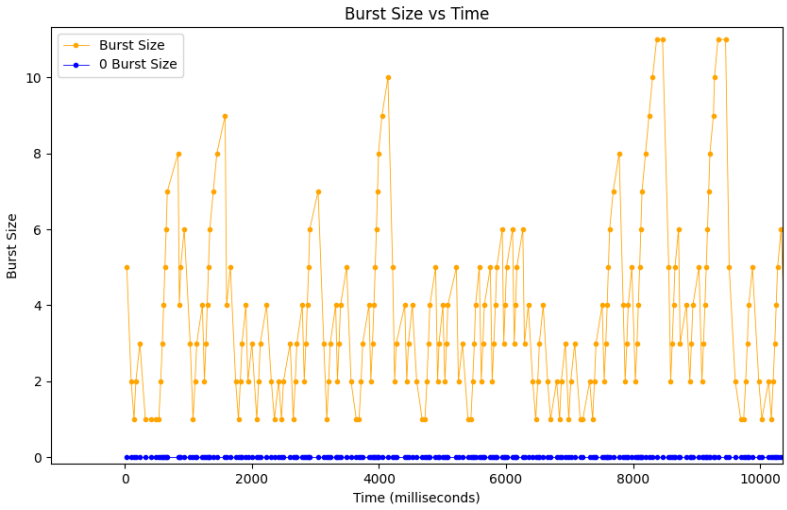


Figure 4: Zoomed in Burst-size vs time; shape for A.I.M.D

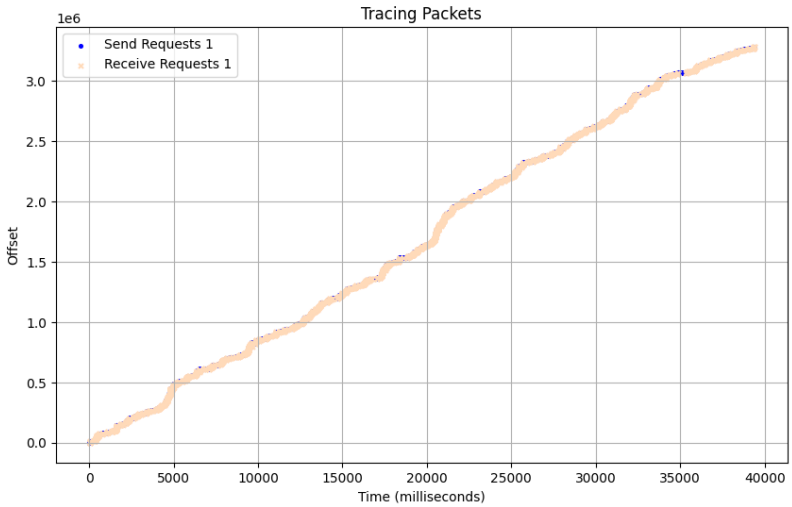


Figure 5: Offset queried vs time. This is monotonic as each burst starts offset queries from 1 and iterates over ones that are currently not in the dictionary.

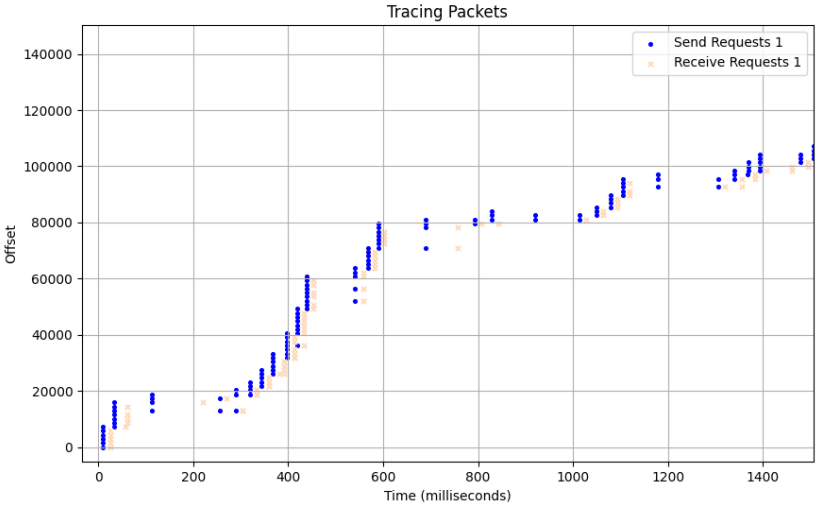


Figure 6: Queries and Received msgs vs time (zoomed in). This shows us the **bursts** that we send requests in. Here in the graph we can observe that when all the requests have not been serviced, the burst size is decreased by 2 times. Further, the **inter-burst times** are also increasing when the burst-size decreases.

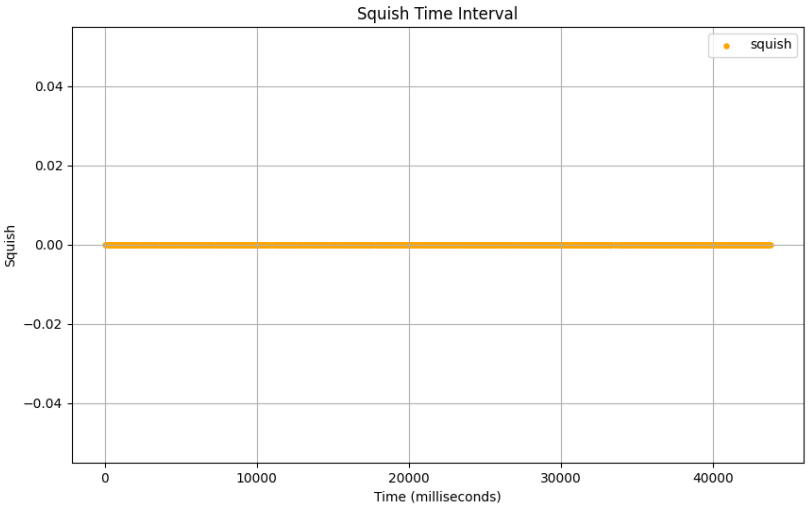


Figure 7: Since there were 0 squishes, squishes vs tine is a flat line at 0

```
Submit: 2021CS10098@pokemon
MD5: 6876a7f852eecf45dc69ad878861b9c5


Result: true
Time: 44868
Penalty: 80
```

Figure 8: Result for running on localhost

```
RTT =  0.0033850669860839844

Submit: 2021CS10098@pokemon
MD5: 39d9ad921281be500ecdf3ea60896b87


Result: true
Time: 36025
Penalty: 47
```

Figure 9: Result for running on vayu (`10.17.7.134`)

7 Team Details

Team Members

- Utkarsh Sharma 2021CS10098
- Priyanshu Ranjan 2021CS10575