COL 334/672 – semester II 2023-24: Assignment 3

Reliable and congestion friendly yet speedy file transfer

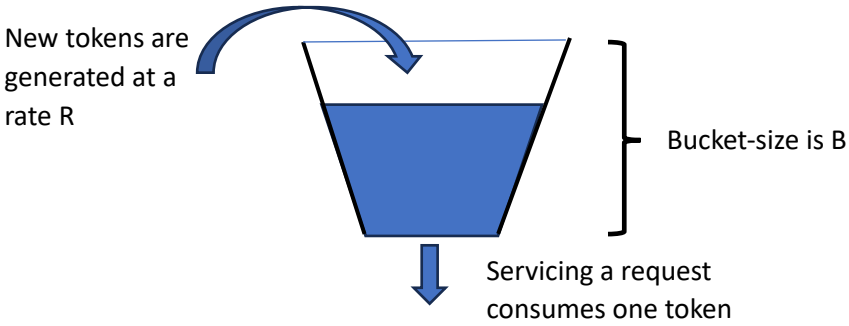
In this assignment, we will try to implement a TCP-like protocol for reliable data transfer, with congestion control-like mechanisms to not overwhelm the network yet obtain high throughput. On popular demand, we may set this up as a tournament as well!

You can do this assignment in teams of two.

Like in the previous assignment, this time we have a UDP server running on vayu.iitd.ac.in. Your job is to implement the client. The client sends requests to the server to receive a certain number of bytes from a certain offset. The server replies to the client with this data, but it makes several checks to decide whether to reply or not:

- To emulate a lossy network, the server randomly decides to not reply every now and then. Internally it samples from a uniformly random distribution and provides a constant packet loss rate.

- To force clients to not send requests too fast, the server implements a leaky bucket filter for each client and replies only if a client's bucket has remaining tokens. A leaky bucket operates on two parameters: a rate and a bucket-size. Tokens are generated at a constant rate, but only a maximum of bucket-size number of tokens can be retained. Whenever the server replies, a token is consumed. If a client makes requests too fast and empties out the bucket (i.e. all tokens have been consumed) then the server does not reply to a new request. After some time, new tokens would get generated (based on token generation rate), and now new client requests would be serviced by the server.



  Leaky buckets are used extensively in the Internet to shape bursty traffic.

- The server further keeps track of requests that could not be serviced because enough tokens were not available. To condone *rude* behavior when clients keep sending requests even though the server is not replying to them, as a penalty the server squishes them by temporarily reducing the rate of generating new tokens, i.e. it would now take longer for clients to receive data.

What you need to do is described next.

1. Receiving data reliably

The server is running on UDP port 9801 on vayu.iitd.ac.in. We will soon set up more servers. Remember that UDP connections do not have a connection establishment phase. You can directly send a request packet to the server by specifying the destination IP address, destination UDP port, source UDP, and constructing the datagram. Each datagram has a format similar to HTTP with header lines terminating in a \n, followed by a blank \n, followed by data.

The very first request a client should send to the server is to know how many bytes to receive. This is done by a simple SendSize command:

SendSize\n

\n

The data portion of this datagram is empty. The server replies with a similar datagram:

Size: [number of bytes]\n

\n

Note that the maximum number of bytes you can expect to receive in this assignment is 15MB.

Next, you can send requests to receive data as follows:

Offset: [offset]\n

NumBytes: [number of bytes]\n

\n

This tells the server how many bytes you are requesting from a particular offset. Offsets are indexed from 0. You can request a maximum of 1448 bytes in one request. Therefore, to now receive the entire data, you will have to send multiple requests, for example, for the first 1448 bytes (*Offset: 0\nNumBytes: 1448\n\n*), then the next (*Offset: 1448\nNumBytes: 1448\n\n*), then the next and so on. The server replies as follows:

Offset: [offset]\n

NumBytes: [number of bytes]\n

\n

NUMBYTES OF DATA

Here, the actual data is sent in the data portion of the datagram.

Remember that packets can get reordered on the network, so you should not expect to receive replies in the same order in which you sent. Further, you should not expect to receive a reply to each request because the server may have decided to drop the request! Or the network may have dropped your request or reply packet. Even SendSize requests could be dropped for that matter. Your implementation will therefore have to robust and clever, to keep sending data requests while not necessarily waiting to receive a reply.

Once you have received and assembled the entire data, you will have to generate an MD5 hash of the data and submit it to the server to check for correctness:

Submit: [entryID@team]\n

MD5: [MD5 hash]\n

\n

You can use a standard library to find the MD5 hash of any sequence of bytes. You will then need to convert each byte to its %02x hexadecimal equivalent (e.g. byte value 0 written as 00, 15 written as 0f, 127 written as 7f, 255 written as ff) and concatenate the hexadecimal equivalent into a string. The string must be in lower case and will look something like:

6bf11f1400185696a3a43a495987756d

Here, there are 32 bytes, written in hexadecimal as a string of 64 characters.

The server replies with:

Result: [true/false]\n

Time: [time taken in ms]\n

Penalty: [penalty]\n

\n

Here, the server reports whether your submitted MD5 hash matched with what it was expecting, the time taken for you to send the results from after the first request from your IP address, and the number of times you sent requests even though your leaky bucket had no tokens left.

An important note: When running in non-tournament mode, you should send a reset request to the server when you start a new connection, i.e. in the very first SendSize request. This will reset your timer in case somebody else was using the same IP address as yours and interacting with the server earlier. This can be done as follows:

SendSize\n

Reset\n

\n

2. Not getting squished

Once you have learned to request and receive data reliably, the next job is to figure out how fast to request for data to not get squished yet not go too slow.

The rate and bucket-size being used by the server are not revealed to you. A key goal of this assignment is for clients to discover when they might be sending requests too fast, and to slow down, to avoid getting squished. You can use TCP's strategy to know when you should slow down – keep track of the

number of replies you expect to be in transit, i.e. requests for which replies are awaited, and if outstanding requests persist for beyond a timeout then assume that request or reply packets were lost. Upon detecting a loss, you can slow down.

If you keep sending requests at a fast rate, remember that the server is tracking how many requests it received when your token bucket was empty. If this exceeds a certain threshold then the server will squish you for about 100 additional requests. The server also helpfully informs you that you have been squished by appending a header line to each subsequent reply it sends during the squish period, so that at least now you should slow down.

Offset: [offset]\n

NumBytes: [number of bytes]\n

Squished\n

\n

NUMBYTES OF DATA

Of course, the goal is to not get squished in the first place. But you should also not send requests too slowly because then you will take forever to receive the entire data. You can consider implementing TCP's AIMD strategy: Upon receiving a reply, increment your sending rate additively; upon detecting a loss, reduce your sending rate multiplicatively. You can choose an initial sending rate based on experiments with vayu or other servers.

3. Logging and visualizations to understand the interaction

It may not be easy to understand what's going on and to debug your algorithms by looking at the text logs alone. Below we explain our own solution implementation and show some graphs to give ideas of what you may want to do as well. Our implementation works as follows:

- We send requests in bursts of a certain burst-size.
- The burst-sizes are maintained using an AIMD approach.
    - Upon each reply that is received, we increment the burst-size by 1/burst-size, so that if all the requests in this burst get serviced, the burst-size would get incremented by 1 for the next burst.
    - If some requests did not receive a reply, we halve the burst-size. Loss detection is done using a timeout: We track when each request was sent, and if a timeout has elapsed without having received a reply, then the request is declared as being lost.
- Timeouts themselves are calculated as a multiple of estimated RTT. The RTT estimate is calculated using EWMA, similar to what TCP does: Upon each reply that is received, the RTT estimate is updated using the value from this new sample.
- Every short quantum of time, almost in a busy-wait manner, we check if all requests in the previous burst have been received or some were declared lost. We then push out the next burst based on the AIMD-updated burst-size.

This is clearly not the best implementation. Perhaps instead of thinking in terms of discrete bursts, we should have used TCP's self-clocking idea itself and simply maintained a congestion window. Maybe you can do a better implementation!

We had the client log each send request it sent (offset, time), the reply received (offset, time), if the client is squished or not (revealed by the server in the replies), and the burst-size maintained by the client. For each run, we saved the trace file and set up a pipeline that could generate graphs such as the ones shown below.

In Figure 1, each point shows the offset and the time for requests sent and replies received. Requests are shown in blue and replies in orange. Due to the scale of the axes, the replies appear to almost overlap with the requests. The client is programmed to run through the entire request space monotonically, 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. This is visible in the graph as the first pass from 0 until approximately 3600 ms, the second pass from 3600ms to approximately 4200ms, etc. Note that the requests all throughout are sent as bursts, it is just that the bursts are not discernible at this scale.

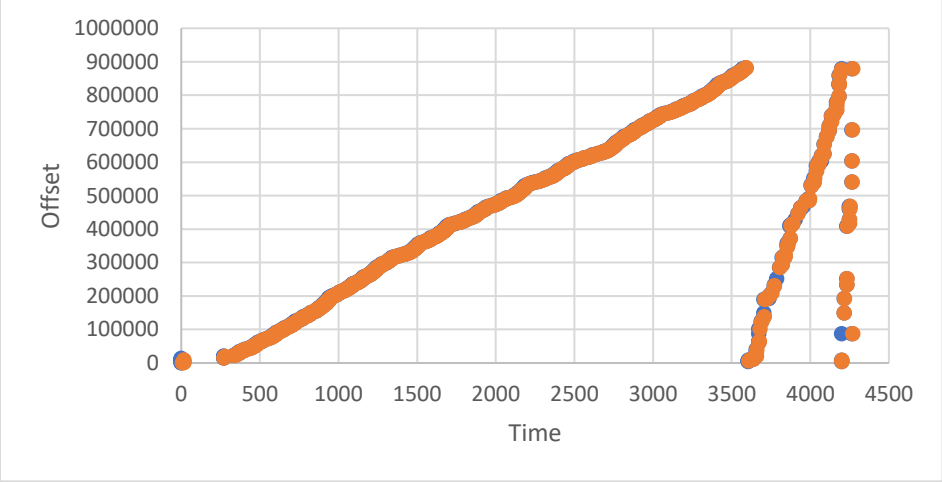


Figure 1: Sequence-number trace

Figure 2 shows a zoomed-in view of this trace for 0 to 500ms. Here, we can clearly see the first burst of 10 requests – hard-coded in our client. Only 4 replies were received and the burst-size was reduced (step 1). The next burst sent at around 270ms contains only 5 requests. This time too, only four replies were received (step 2). The burst-size is therefore reduced to 2 and replies were received to both requests (step 3). Now, the burst-size is incremented to 3 and all three replies were received (step 4). The burst-size now is incremented to 4 but only 3 replies were received (step 5). And so on.

Figure 2: Zoomed in sequence-number trace

Notice that the spacing between the first and second burst is large, it is smaller between the second and third burst, and even smaller after that. This spacing starts with a large conservative value and is successively reduced to approach an ideal gap wherein almost all data from the previous burst has been received and now a new burst can be issued. In retrospect, we feel this was not a good decision, and you may want to simply use a self-clocking method.

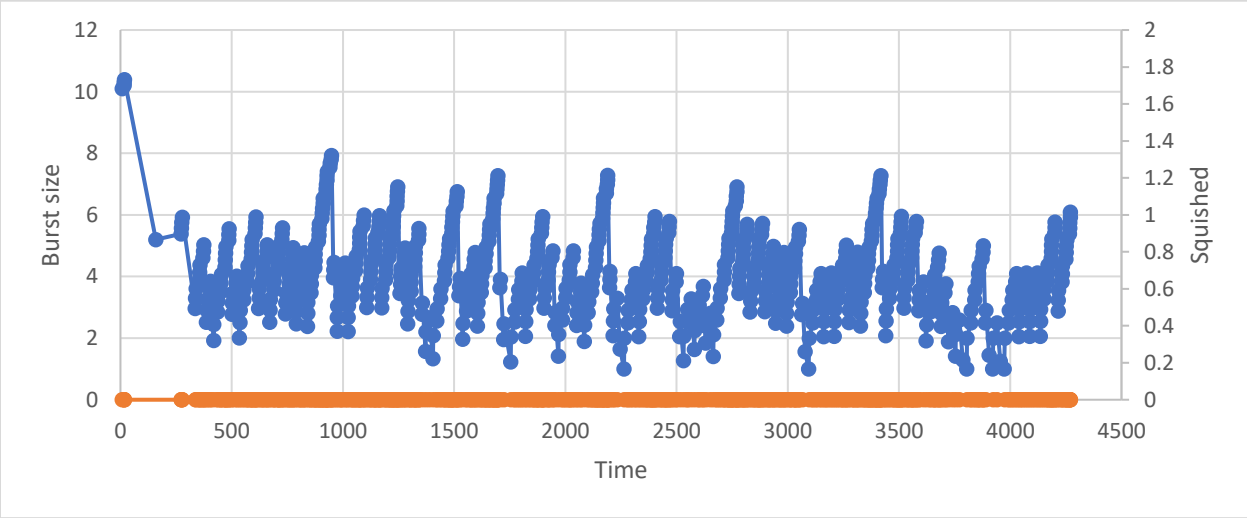


Figure 3: Burst-size for an AIMD behaving client

Figure 3 shows the burst-size over time. Here the classic AIMD saw-tooth behavior can be clearly seen. Note that the burst-size shown here is a floating point value; the actual number of requests sent in a burst is the rounded value of this variable.

In contrast, Figure 4 shows the burst-size for a client that uses a fixed burst size of 10. This client gets squished at around 800ms. Although the server squishes a client for about 100 requests only, in this case, by the time the squish period for this client expires, it has caused additional penalties because it did not slow down and therefore it gets squished again.

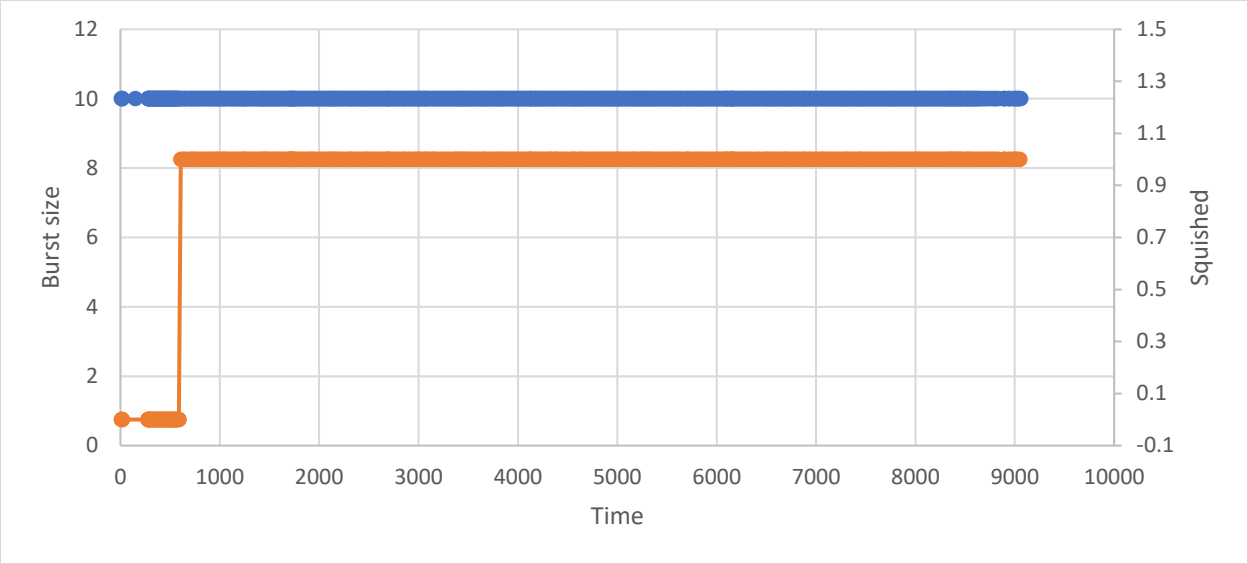


Figure 4: Burst-size for a constant burst-size client

4. A clever hack

Since we have revealed that the server implements a leaky bucket filter, and the assignment is meant to be executed in a known environment (i.e. in the IITD campus network, with communication to vayu or another similar server), you can conduct experiments to determine an optimal rate at which to send so that you do not get squished and are yet able to download the file quickly. In fact, a good choice of burst-size and wait-time for a trivial fixed burst-size client we implement actually does better than our way more complex AIMD client!

5. But not generic

This clever hack will however not work in a dynamic environment where the available bandwidth may change. Sending at a fixed rate would mean that the client can get squished if the server's rate lowers to below this fixed rate, and correspondingly, the client may not be able to take advantage if the server' rate increases.

We have therefore configured another server instance that does not hold the leaky bucket rate parameter constant. Every two seconds or so, the server increases its rate and then decreases, to emulate a network that gets uncongested and congested in an alternating manner. Figure 5 shows the burst-size trace for our AIMD client. The client is able to successfully adjust to larger burst-sizes during periods when the server allows a higher rate, and reduces to lower burst sizes during periods of lower rate. The client does not get squished even once.

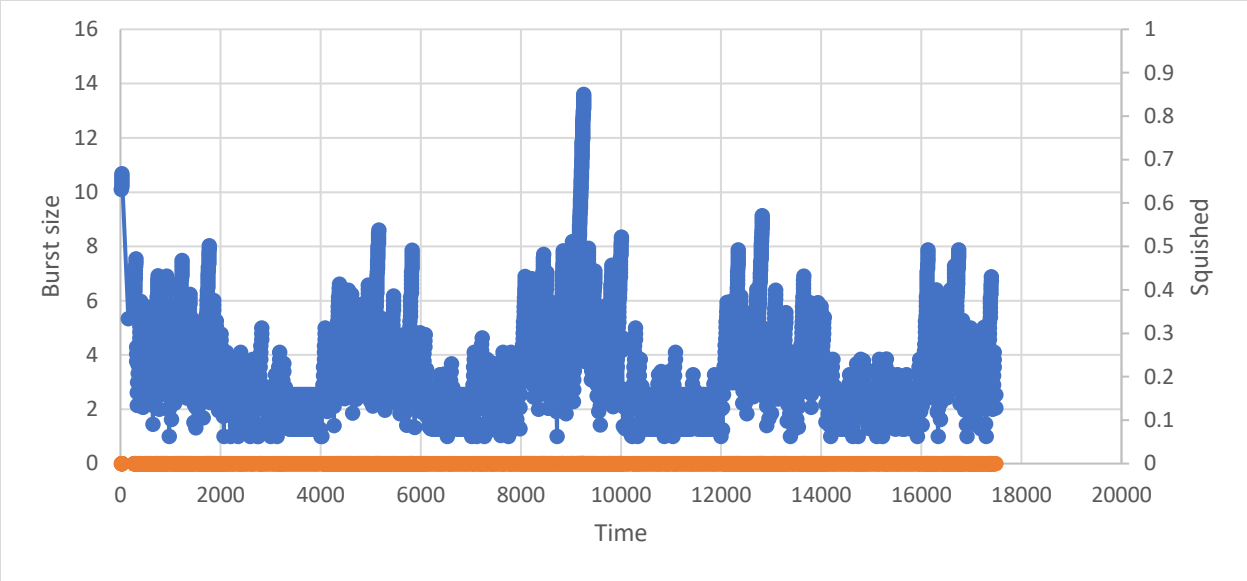


Figure 5: Burst-size in a variable rate server

On the other hand, Figure 6 shows a burst-size trace for our trivial client implementation that maintains a constant burst-size. Here, the client is able to recover from the squish period periodically especially when it coincides with a higher rate period at the server when new tokens are generated more quickly, but gets squished soon after when the server reduces the rate. Our AIMD client is able to beat the constant burst-size client in such variable rate settings because it is able to adjust to the changing available bandwidth.

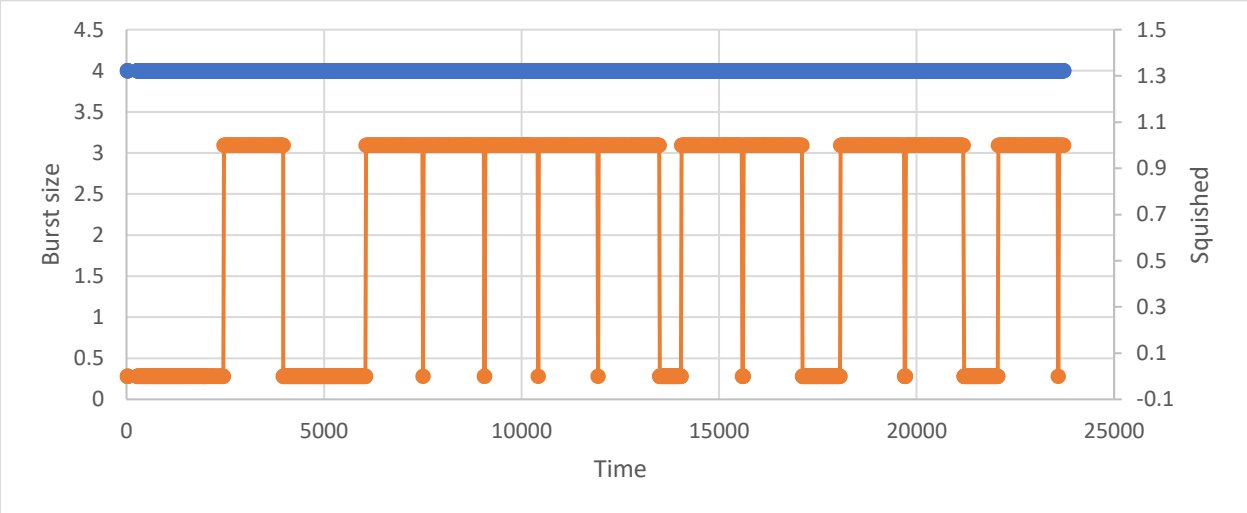


Figure 6: Burst-size trace for a constant burst-size client

6. What to submit

You should do this assignment step by step. We will also have intermediate milestones that you should meet by the indicated deadline.

Milestone 1: Due Oct 15th

- Start with demonstrating that you can send and receive UDP packets in accordance with the protocol specified here.
- Next, show that you can receive data reliably. Send your requests slowly; note that some will still not receive a reply since the server emulates a loss rate or the network may have dropped the request or reply packet; assemble the received data and send additional requests for data that was not received. Verify the MD5 hash from the server.
- Create graphs such as the ones shown above.

Milestone 2: Due Oct 22nd

- Now, implement your client for the case where the server maintains a constant rate for its leaky bucket parameters. Report the time taken and penalty incurred in receiving the entire data. Use graphs to show squish periods, congestion window sizes (if this is what you choose to implement), sequence numbers, etc. Report any interesting methods you used to determine an optimal requesting rate and other parameters for your implementation.

Milestone 3: Due Oct 29th

- Finally, generalize your client to be able to handle variable rate scenarios. Report the timing, explain your methods, and show graphs that can help understand this better.

For each milestone, submit your code and a report covering all the corresponding aspects. Your TA mentors will also conduct a viva. You should be able to demonstrate your work through a running system and explain your methods.

We are operating two servers on vayu which you can use for testing and calibration:

- Constant rate server, running on UDP port 9801
- Variable rate server, running on UDP port 9802

This time, we will also provide you with a compiled server Java bytecode that you can run locally so that you can debug your code more easily. The server is written in java and compiled on the latest JDK, Java SE 20. Install the JDK, download the following files to a local folder, and run using the following in the same folder:

https://www.cse.iitd.ernet.in/~aseth/temp/col334/UDPServer.class

https://www.cse.iitd.ernet.in/~aseth/temp/col334/ClientConn.class

https://www.cse.iitd.ernet.in/~aseth/temp/col334/big.txt.1

java UDPServer [port number] [filename] [maxlines] [variablerate] [tournament] [verbose]

For example:

- java UDPServer 9801 big.txt.1 10000 constantrate notournament verbose

  This starts the server on port 9801. The server reads 10000 lines from a file big.txt – this is the data that the server will send to clients. The server uses a constant rate, does not run in a tournament mode, and logs interactions with the clients in a verbose manner.

- java UDPServer 9802 big.txt.1 10000 variablerate notournament noverbose

  The server here uses a variable rate, does not run in a tournament mode, and does not log detailed interactions with clients.

You can write your client programme and run it on the same machine to connect to the server, e.g.

- java UDPClient 127.0.0.1 9801

You can of course write the client programme in any language.

The logs produced by the server contain information such as the following:

- Connection initiation:

  Received: 127.0.0.1 – SendSize

  Sent size: 127.0.0.1, size: 3289631

  This is a log of the server having received a SendSize command, to which it replied with sending the size.

- Data request and reply:

  Received: 127.0.0.1 - Offset: 0

  Received: 127.0.0.1 - NumBytes: 1448

  Tokens remaining: 127.0.0.1, tokens: 4, cumulPenalty: 0, runningPenalty: 0

  Sent data: 127.0.0.1, offset: 0, size: 1448, squished: false

  This is a log of the server having received a data request to send 1448 bytes from offset 0. The server has 4 tokens remaining and the client has not incurred any penalty so far. The server therefore replies with data and sends 1448 bytes from offset 0. The log also indicates that the client is not squished.

The cumulPenalty value is the total number of penalties incurred so far in the entire session, and runningPenalty is the number of penalties since the last squish release time. The penalties are incremented by 1 each time a request is received but there are no remaining tokens.

- Data request and skipped reply:

  Received: 127.0.0.1 - Offset: 432952

  Received: 127.0.0.1 - NumBytes: 1448

  Tokens remaining: 127.0.0.1, tokens: 0, cumulPenalty: 10, runningPenalty: 10

  Tokens exhausted: 127.0.0.1, penalty: 11, squishTime: 0

  Skipped request: 127.0.0.1, offset: 432952

  In contrast with the previous log, this shows the server received a request for 1448 bytes from offset 432952, but the tokens were exhausted and in fact had been exhausted 10 times earlier too, to have incurred a penalty of 10 so far. The server therefore skips sending a reply and increments the penalty to 11.

- Final result

  Received: 127.0.0.1 - Submit: aseth@col334-672

  Received: 127.0.0.1 - MD5: 8ba62ddde9968a075c10ba5c958bb1fa

  Sent success: 127.0.0.1, time taken: 18645, cumulPenalty: 378, runningPenalty: 32

  Upon having received the entire file, the client makes a submission. The server logs having received the submit request and the MD5 hash. Since the hash matches, the server replies with a success and also records the time taken (in milliseconds), the total penalty incurred, and the penalty incurred since the last time the client was squished.

With this information, you should be able to begin your client implementation locally and then move to interacting with the server running on vayu.