COL 334 Assignment-2

Rajat Bhardwaj
2020CS50436

September 2022

1 Part 1

1.1 Server

The functioning of each ports is as follows. Here n = number of clients

- 1 TCP port (server_TCP) for distributing the chunks to client. Firstly the splitfile(filename) function encodes the file and splits the file into x 1kb chunks. Then each client receive x / n chunks (+1 more chunk if x % n != 0 for some clients) via this port.
- 1 UDP (server_UDP)port to save the udp addresses of the client side. When I declared some UDP ports on the client side for receiving the broadcast request I used this server udp port to check the working of these udp ports by sending 'HELLO!' to the server. I also saved the address of the udp ports of the client on the server side because I needed to send the broadcast from the server to these ports.
- n TCP ports (server_broadcast_tcp[]) to accept the chunks which were requested by the server (broadcasted) from the clients who had those requested chunks. The accept_tcp(id) function runs in parallel for id = 1 to n (threading) and keeps listening to the client. The ith port listens to the ith client. As soon as all the clients receive all the chunks all these tcp ports are closed.
- n UDP ports (server_broadcast_sock[]) to accept the request of a chunk from the clients. The ith UDP port listens to the ith client in the function send_reqeuested_chunk(id) that runs in parallely for id = 1 to n (threading) . As soon as it receives a request of a chunk it checks the cache if this chunk id is present or not, if it isn't present it calls the broadcast(chunk_id, client_id) function
- n UDP ports (server_UDP_random_broadcast[]) to broadcast request in broadcast() function. The ith UDP port sends request to the ith client.
- n TCP ports to send the requested chunk to the client (after acquiring the chunk from the cache or other client) in the function send_reqeuested_chunk(id). These TCP ports are opened in the function itself, sends the chunk and closed. Maximum n such ports can be opened at once because send_reqeuested_chunk(id) function is running in parallel for id = 1 to n.

So the total number of TCP ports are 1 + n + n = 2n + 1
The number of UDP ports are 1 + n + n = 2n + 1

1.2 Client

The number of ports and their functioning for each clients are

- 1 TCP port (clients[id][0]) to accept the chunks which are distributed (disjointedly) in the receive_chunk(client , id) function. The chunks are stored in the list client_chunks[id]. The client_chunks is a list of list.

Each index representing the chunks with each client. ith client can access only the ith index of the client_chunks list. Each chunk is in the form of a 3-tuple - (chunk_id , chunk_size, data). The data is encoded initially and it is always of the size 1024. It is decoded in the end after combining all the chunks.

- 1 UDP port (client_UDP_random[id]) to request the server for the missing chunk in the request_chunk(chunk_id , id). This function requests the $chunk\_id^{th}$ chunk from the server for the requesting client number = id.
- 1 UDP port (clients[id][1]) to receive request (broadcast) from the server in the function handle_broadcast(id) function.
- 1 TCP port (client_TCP_random) to send the requested chunk to the server, if it is present with the client in the handle_broadcast(id) function. This socket is opened inside the function whenever the chunk is to be sent to the server, the port is selected randomly. After sending the chunk the socket is closed.
- 1 TCP port to accept the requested chunk from the server in the request_chunk(chunk_id , id) function.

So the total number of TCP ports are 1 + 1 + 1 = 3 per client
The number of UDP ports are 1 + 1 = 2 per client

1.3 Packet loss handling

1.3.1 client side

In the function request_chunk(chunk_id, id). It sockettimeout exception is caught and the loop is continued. If the packet is received the loop is broken.

1.3.2 server side

In the function send_requested_chunk(chunk_id, id) , timeout = time.time() + 5 is used, i.e if the time() becomes equal to timeout, the server will do a broadcast again, otherwise, the loop is broken

2 Part 2

2.1 server

- 1 UDP port server_UDP is used to distribute the file chunks to the clients in the distribute_file_to_clients(filename) similar to part 1.
- n TCP port server_TCP_random for broadcasting the message to the clients in the broadcast(request , udp_client_port) function. This tcp port is opened when the function is called, after the broadcast the socket is closed. At once only n ports can remain active as the broadcast() function runs on n threads.
- n UDP ports server_broadcast_sock[]for receiving the requested chunks in the function accept_udp(id).
- n TCP ports server_broadcast_tcp[] for receiving the chunk_id request from the clients in the function send_reqeuested_chunk(id)
- n udp ports server_UDP_random for sending the requested chunks from the clients in the function send_reqeuested_chunk(id)

So the total number of TCP ports used by the server are n + n = 2n
The number of UDP ports used by the server are 1 + n + n = 2n + 1

2.2 client

- 1 UDP port clients[id][1] is used for 2 functions. Firstly the chunks distributed by the server are received on this port in the receive_chunk(client, id) function. The second use of this port is to receive the requested chunk from the server in the request_chunk(chunk_id , id) function.
- 1 tcp port client_binded_tcp[id] for receiving the server broadcast request in the in the handle_broadcast(id).
- 1 udp port client_UDP_random to send the requested chunk to the server in the handle_broadcast(id) function.
- 1 tcp port client_TCP_random to send request to the server for missing chunk_id in the request_chunk(chunk_id , id)

So the total number of TCP ports used by each client are 1 + 1 = 2 per client
The number of UDP ports used by each client are 1 + 1 = 2 per client

2.3 Packet loss handling

2.3.1 server side

Initial distribution also requires packet loss handling.In the send_chunk((chunk , conn , chunk_id)) function After a chunk is sent by the server, the server waits for an acknowledgement for the same from the client. If the acknowledgement received is corrupted or it is not received (socket timeout) the server re-sends the same chunk.

The chunks which are sent to the server by the client on request(broadcast) can also drop. This is handled in the function send_requested_chunks() by timeout.time() = 5. i e. if the time.time() becomes equal to timeout.time() then the server rebroadcasts the request for the same chunk else if the chunk is added to the cache before the timeout the loop is broken.

2.3.2 client side

In the initial distribution the client sends an acknowledgement for the id of the packet received.

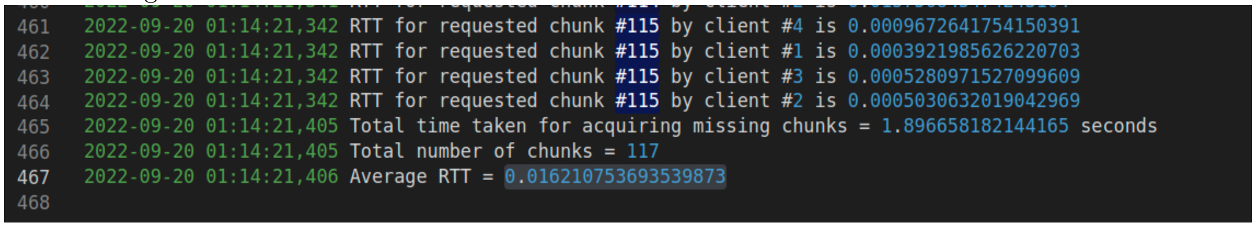
In the request_chunk() function If the requested chunk is not received by the client and the socket timeout exception is handled by re-sending the chunk request to the server.

3 Analysis

3.1 Average RTT for each chunk

3.1.1 Part 1

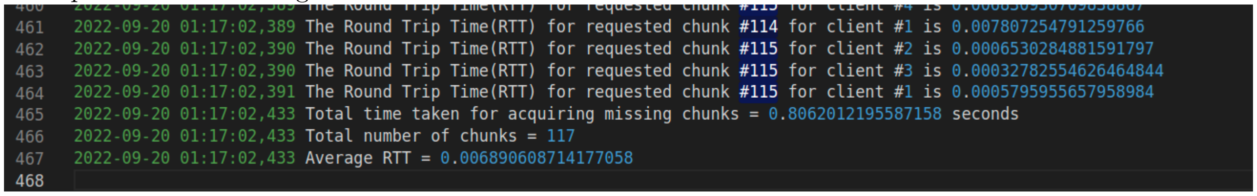
For the part 1 the average RTT for each chunk is 0.016210753693539873 seconds. The RTTs are reported in the std.log file.



```
2022-09-20 01:14:21,342 RTT for requested chunk #115 by client #4 is 0.0009672641754150391
2022-09-20 01:14:21,342 RTT for requested chunk #115 by client #1 is 0.0003921985626220703
2022-09-20 01:14:21,342 RTT for requested chunk #115 by client #3 is 0.0005280971527099609
2022-09-20 01:14:21,342 RTT for requested chunk #115 by client #2 is 0.0005030632019042969
2022-09-20 01:14:21,405 Total time taken for acquiring missing chunks = 1.896658182144165 seconds
2022-09-20 01:14:21,405 Total number of chunks = 117
2022-09-20 01:14:21,406 Average RTT = 0.016210753693539873
```

3.1.2 Part 2

For the part 2 the average RTT for each chunk is 0.006890608714177058 seconds. The RTTs for all chunks are reported in the std.log file



```
2022-09-20 01:17:02,389 The Round Trip Time(RTT) for requested chunk #114 for client #1 is 0.007807254791259766
2022-09-20 01:17:02,390 The Round Trip Time(RTT) for requested chunk #115 for client #2 is 0.0006530284881591797
2022-09-20 01:17:02,390 The Round Trip Time(RTT) for requested chunk #115 for client #3 is 0.00032782554626464844
2022-09-20 01:17:02,391 The Round Trip Time(RTT) for requested chunk #115 for client #1 is 0.0005795955657958984
2022-09-20 01:17:02,433 Total time taken for acquiring missing chunks = 0.8062012195587158 seconds
2022-09-20 01:17:02,433 Total number of chunks = 117
2022-09-20 01:17:02,433 Average RTT = 0.006890608714177058
```

3.2 Which RTT is higher?

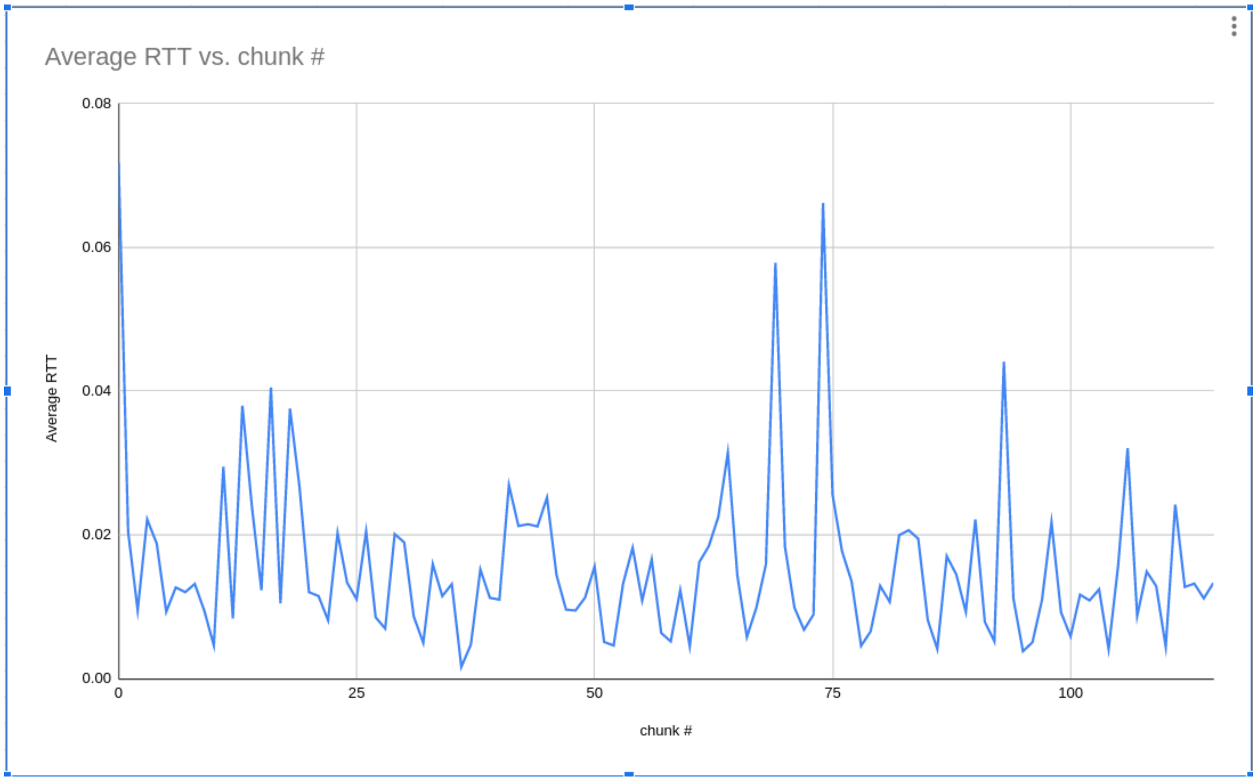
We know that UDP takes lesser time than tcp.

The averge RTT of part1 is 2.35258659517 times higher than the RTT of the second part. This was expected as in the part 1 the chunks are transferred using TCP where as the requests requests which are smaller in size (100 times smaller than the chunk) are transferred by the UDP. Therefore the transfer of chunks dominate the time taken in the RTT.

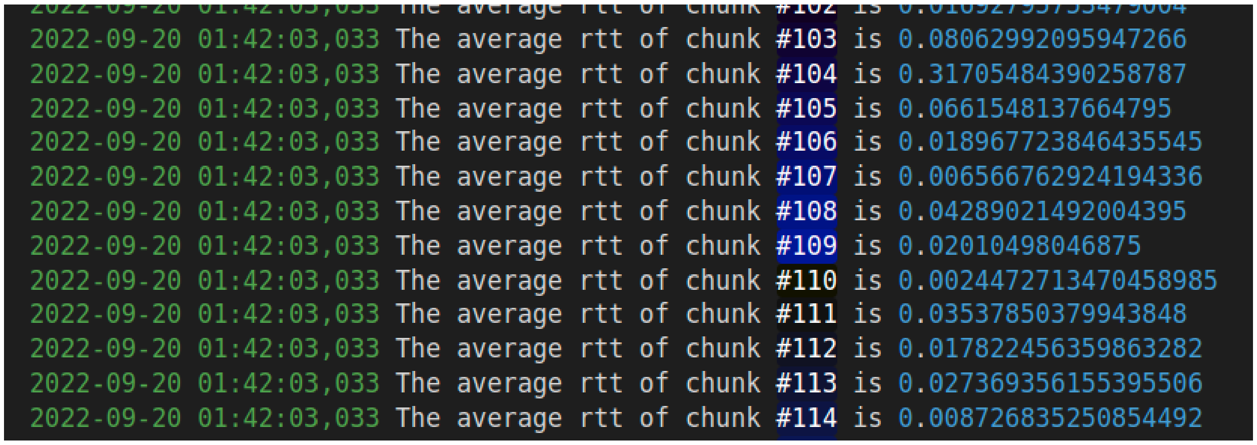
In the 2nd part the transfer of chunks happened via UDP therefore it took lesser time as compared to part 1 where the transfer of chunks happened via tcp.

3.3 the average RTT for each chunk across all clients

3.3.1 Part 1



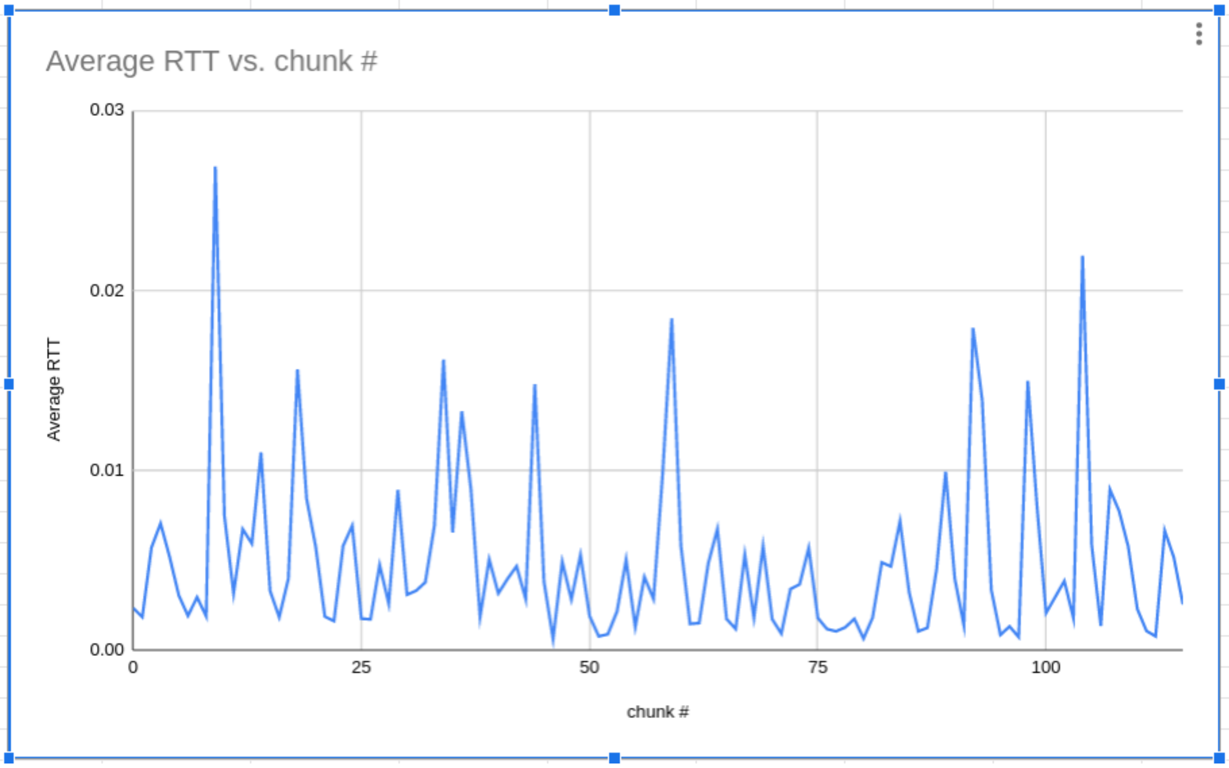
The average RTT for each chunk across all clients is reported in the std.log file.



```
2022-09-20 01:42:03,033 The average rtt of chunk #103 is 0.08062992095947266
2022-09-20 01:42:03,033 The average rtt of chunk #104 is 0.31705484390258787
2022-09-20 01:42:03,033 The average rtt of chunk #105 is 0.0661548137664795
2022-09-20 01:42:03,033 The average rtt of chunk #106 is 0.018967723846435545
2022-09-20 01:42:03,033 The average rtt of chunk #107 is 0.006566762924194336
2022-09-20 01:42:03,033 The average rtt of chunk #108 is 0.04289021492004395
2022-09-20 01:42:03,033 The average rtt of chunk #109 is 0.02010498046875
2022-09-20 01:42:03,033 The average rtt of chunk #110 is 0.0024472713470458985
2022-09-20 01:42:03,033 The average rtt of chunk #111 is 0.03537850379943848
2022-09-20 01:42:03,033 The average rtt of chunk #112 is 0.017822456359863282
2022-09-20 01:42:03,033 The average rtt of chunk #113 is 0.027369356155395506
2022-09-20 01:42:03,033 The average rtt of chunk #114 is 0.008726835250854492
```

3.3.2 Part 2

The average RTT for each chunk across all clients is reported in the std.log file.

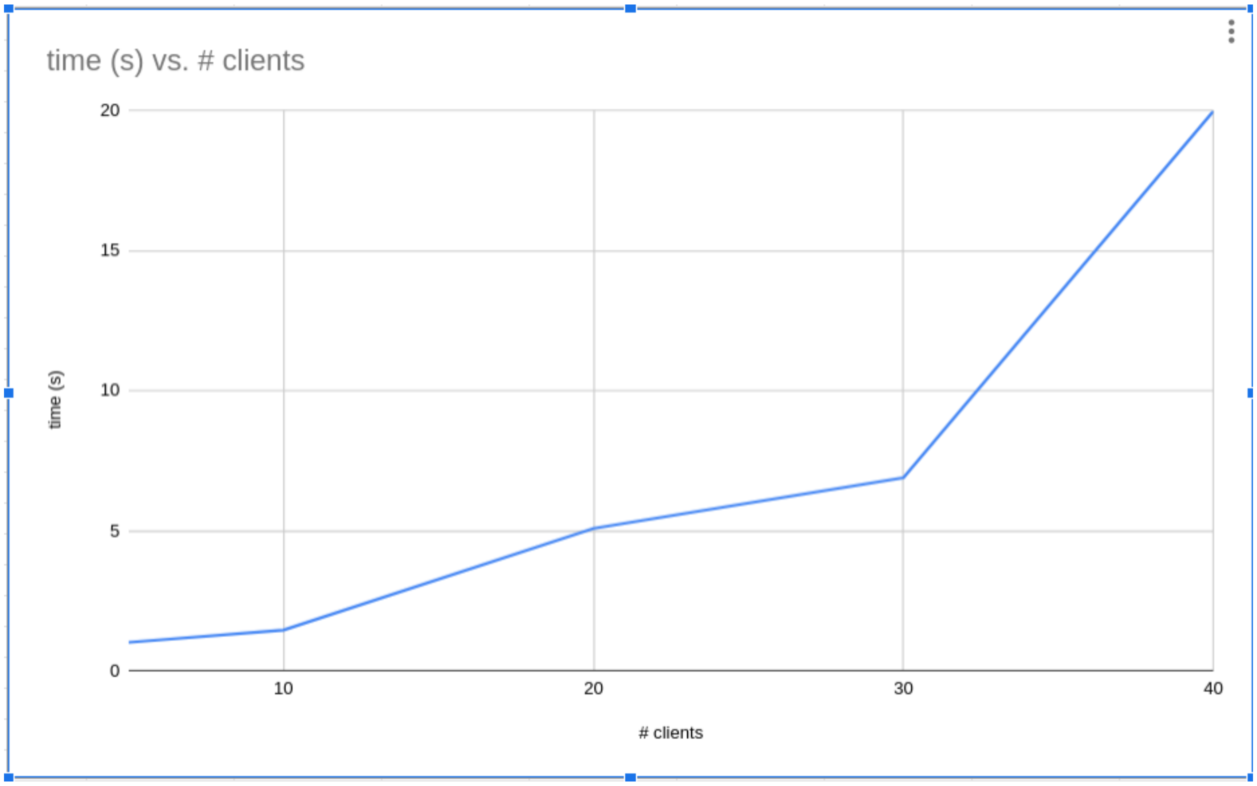


3.3.3 Chunks with higher RTT

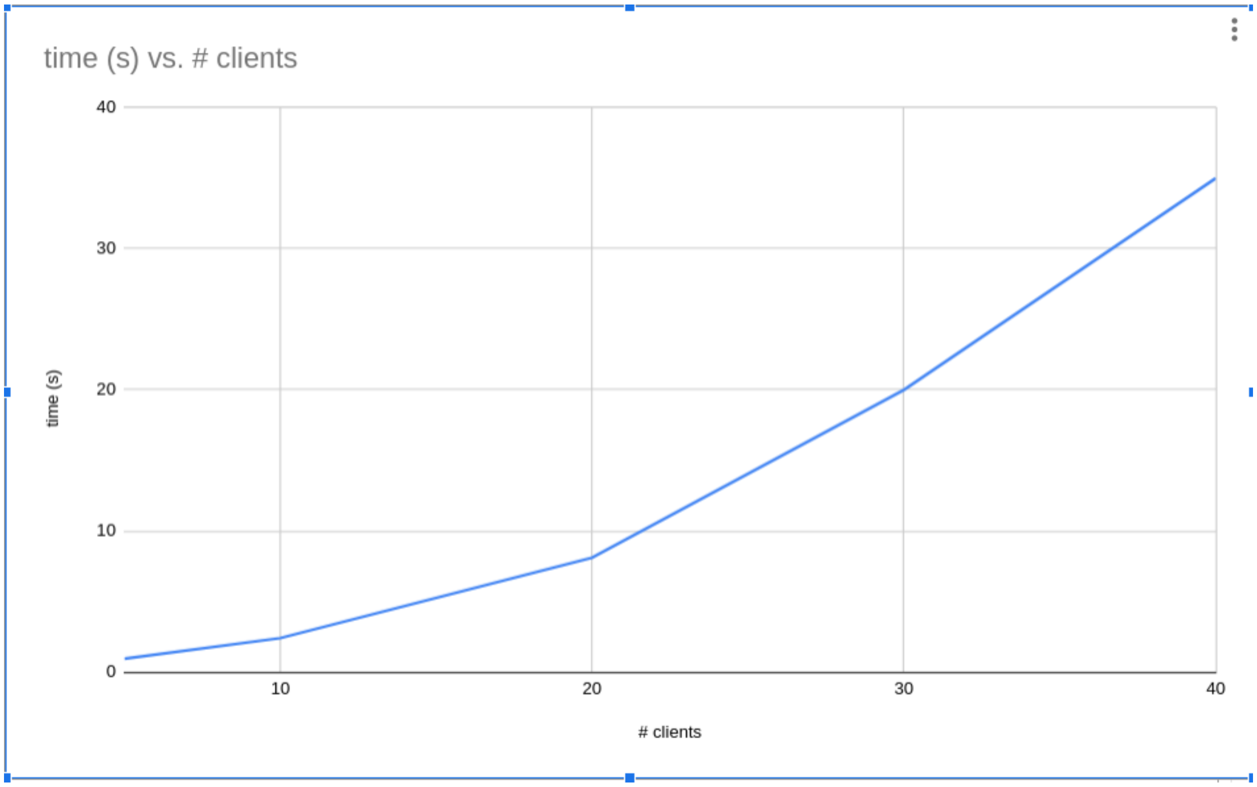
There are some chunks whose average RTT is significantly higher than the rest of the chunks in both parts. These chunks must have been dropped and re-requested.

3.4 Run the simulations for n = 5, 10, 20 and 30 , 40

3.4.1 Part 1



3.4.2 Part 2



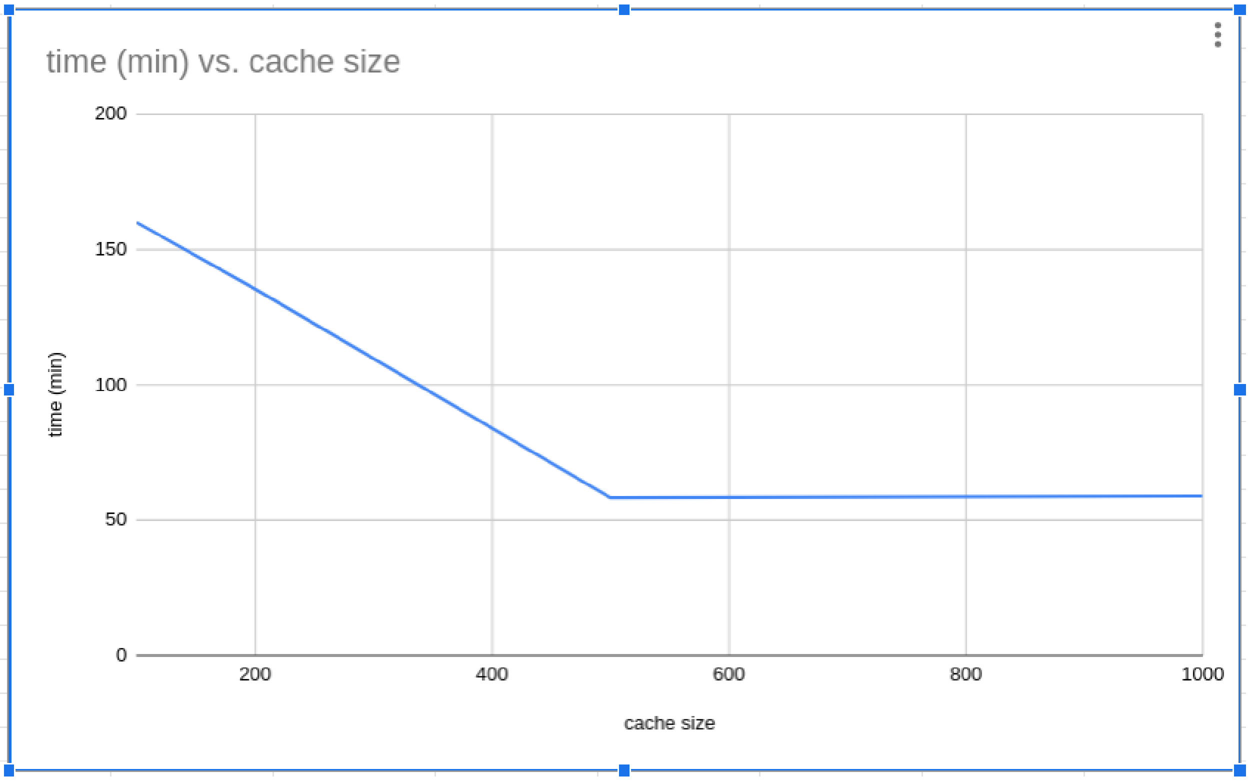
3.4.3 Observation

I observed that the network works fine for a specific number of clients but when the number of clients become 30 the time taken by the code to finish becomes significantly large. This might be due to the congestion at the requested chunks receiving ports at the server side (because in the end more and more clients send the requested(broadcasted) chunk to the server. I also observed that the RTT for each chunk grows with time the reason is same.

3.5 Large file

For large file and n=5 the code takes about 20 mins to run. For n = 30 the approx time to run is around 160 min.

cache size	time (min)
100	160.0310523
200	135.4002476
500	58.1982839
1000	58.86683582



Therefore increasing cache

3.6 Random vs Sequential

3.6.1 part 1

The random request takes 4.608035087585449 seconds total whereas the sequential request takes 2.046358585357666 seconds.

To replicate, set the random_request = True for sending the request randomly.

3.6.2 part 2

The random request takes 1.8069736957550049 seconds whereas the sequential request takes 0.6705217361450195 seconds

3.6.3 observation

The random request takes more time. This was expected because, during the sequential request there is a high probability that the chunk is present in the cache. In random request there is a really low probability because the requests for the chunks for each client is very different where as when we request sequentially, the client requests a chunk which was just requested by another client a while ago and is present in the cache of the server.

4 Food for Thought

4.1 advantage this network has over a traditional file distribution

One of the advantage of this network is that the server doesn't need to store all the files. The server can send the file to the client by finding out which other client has the file and requesting the client to send the file to the server and then the server can send the file to the requesting client and delete the file this having more storage capacity. The recent files can be stored in the cache to make it available to other clients in case they request the same file.

4.2 advantage of this network over a traditional P2P network

The advantage can be security. When a peer asks for a file from other peer the other peer may send a bugged file. In PSP network the server can check whether the file is virus free or not.

4.3 multiple files across the clients

The header of each chunk must contain the ID of the file which should be unique for each file. Header can be of the form CHUNK_ID ..... FILE_ID. Where chunk ID is a unique ID for all the chunks of that file and the file ID is unique for each file. This header can be used to identify the chunk requested by the client. Each client can store a nested dictionary where the main dictionary tells the FILE id with the client and at each file ID the nested dictionary contains the ID of the chunk with the client and the key to that is the location of the chunk.