PSP FILE DISTRIBUTION NETWORK

Using Socket Programming in Python

- At the end of running both client server files for part-1, for each client, a text file is created that prints the MD5 sum of its chunks collected.
- When a client receives any chunk, it stores it in its database in the form of a text file for each chunk.

1. Working of the PSP network:

Part 1: Using UDP for control messages and TCP for data transfer

- **Designing the Code: Simple PSP network with 5 clients:**

  The networks has the following components with work sequentially:

  (1) **Server distributing files to the clients**:
  - In this part, client has to be run first.
  - To make the chunks of equal size of 1kB (for easy processing), the textfile was divided into chunks of 1000 characters(bytes) each, and the last chunk was appended with characters # to make its size equal to 1000.
  - Ports were alloted to server and clients for data handling.
  - n threads are created where each thread is alloted to a particular port of the server that is dedicated to a particular client. Similarly n clients threads also work simultaenously.
  - Initially, server sends to clients how many chunks is present in total. In the consequent chunks, server sends the data by appending its client ID to the front (10 bytes) to the clients. It waits for timeout 1s to recieve acknowledgement "ACK" from the client, else it resends it.
  - To keep an account of acknowledgements recieved, a list of "ACKS" is maintained, where each element of the list is alloted to a port. This ack is recieved from another socket of the server side, which on recieving "ACK" makes the element as "ACK". When this is made the server's port proceeds sending the next chunk till all the chunks are send, after which the threads merge.
  - On the client side, the socket listens to requests from server for connections. The client on recieving data sends an acknowledgement "ACK" to the server.

  (2) **Client requesting missing files from server:**
  - Clients then check their database to find which chunks are missing. For each missing chunk, Clients send a request for the chunk via UDP to a random port of the server. It then waits for the "ACK" for timeout = 2s, after which it again resends it.

– The server's port (if it is free) on recieving the request sends an "ACK" to the client via TCP. This acknowledgement is handled in the same way by maintaining a list of "ACKS" designated for each client. It also adds the requests to a database(QUEUE) maintaining the requests with chunk ID and client ID

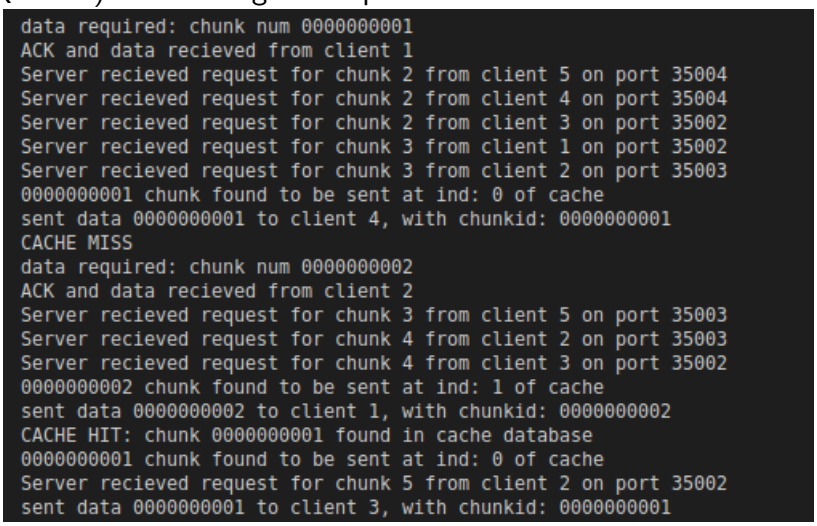


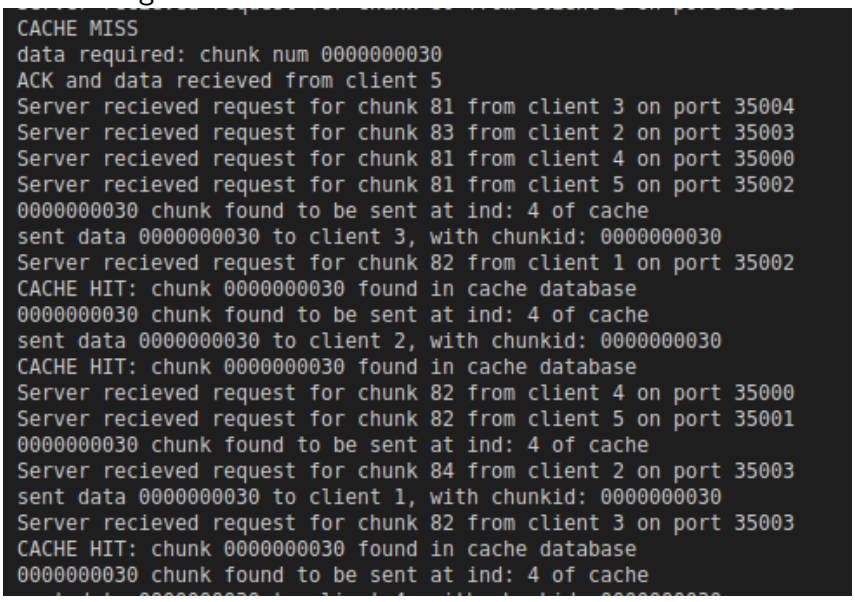
Fig: Server receiving client requests:

Sending requests via UDP to random ports often result in packet loss due to congestion, so the client resends it if it doesn't recieves the acknowledgement within timeout

(3) **Server checking in the Cache:**

Now, the server checks for the requests in the QUEUE. It pops the request, finds for the chunk in the cache, if present, it sends the data to the client that requires it. Else it broadcasts for the same.

Fig: Server CACHE HITS and CACHE MISSES:



(4) **On Cache Miss, server broadcasts for file from client:**

– Here again threads run in parallel requesting the file for the chunk from each client via UDP. It similarly maintains a queue for ACK.

– Client on recieving the broadcast keeps it in a queue. Now the client checks if the data chunk is present in its database. If so, the client sends "ACK" appended

to the data, else it sends "NACK" to the server's port via TCP. All this work in separate threads, so separate elements of a list work as queue for each thread for non-confusion.

- The server then recieves the ACK/NACK via its TCP sockets, and if the data is recieved along with ACK, then it appends it to the cache and sends it back to the client which requires it.

Fig: clients receiving server broadcasts

```
broadcast recieved for chunk68 in client 1
broadcast recieved for chunk68 in client 3
broadcast recieved for chunk68 in client 4
broadcast recieved for chunk68 in client 5
recieved data of chunk 68 for client 2
broadcast recieved for chunk66 in client 1
broadcast recieved for chunk66 in client 2
broadcast recieved for chunk66 in client 3
broadcast recieved for chunk66 in client 5
recieved data of chunk 66 for client 4
recieved data of chunk 66 for client 3
recieved data of chunk 66 for client 5
broadcast recieved for chunk67 in client 1
broadcast recieved for chunk67 in client 2
broadcast recieved for chunk67 in client 5
broadcast recieved for chunk67 in client 3
```

- **RTT (Round Trip Time):**
  - I have taken wait time *(timeout) of 2 seconds* for the client to send request and recieve acknowledgement.
  - Difference in time between sending requests and recieving acknowledgement : 2.41 seconds (on average) But this includes the sleep time (2 seconds), so we need to subtract that time 2.41 seconds.
    So RTT = 2.41-2 = **0.41 seconds** *(on average)*
  - The average RTT for each chunk is coming in the range: 0.38-0.48 seconds
  - But, the RTT for chunk 10, chunk 23 etc.. came in the range of 1.23-1.25 seconds. Also, initial chunks had larger rtt value, as they had to be broadcasted (cache miss). This meant **the packet was dropped by UDP due to congestion at server's port, as it doesn't provide relaible data transfer**. So, it was again resent and then ack was recieved. So it included extra time for resending the data

Fig: RTTs for client 1

client1RTT.txt

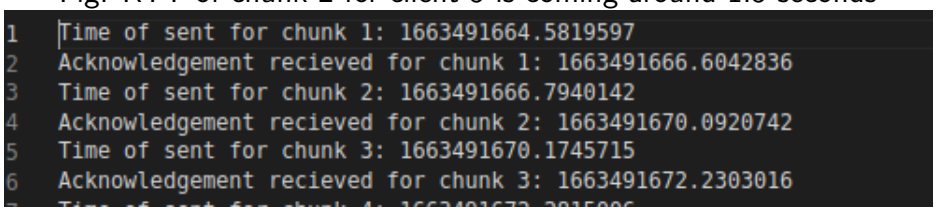
```
1  Time of sent for chunk 2: 1663488039.9495838
2  Acknowledgement recieved for chunk 2: 1663488042.0458274
3  Time of sent for chunk 3: 1663488042.1127462
4  Acknowledgement recieved for chunk 3: 1663488044.2489543
5  Time of sent for chunk 4: 1663488044.3529055
6  Acknowledgement recieved for chunk 4: 1663488046.4658654
7  Time of sent for chunk 5: 1663488046.5739238
8  Acknowledgement recieved for chunk 5: 1663488048.6726236
9  Time of sent for chunk 7: 1663488048.72409
10 Acknowledgement recieved for chunk 7: 1663488050.82553
11 Time of sent for chunk 8: 1663488050.8429751
12 Acknowledgement recieved for chunk 8: 1663488052.9435105
13 Time of sent for chunk 9: 1663488053.0779278
14 Acknowledgement recieved for chunk 9: 1663488055.0995014
15 Time of sent for chunk 10: 1663488055.2066681
16 Acknowledgement recieved for chunk 10: 1663488057.2372367
```

Fig: RTTs for client 3

client3RTT.txt

```
1  Time of sent for chunk 1: 1663488039.9497788
2  Acknowledgement recieved for chunk 1: 1663488042.045911
3  Time of sent for chunk 2: 1663488042.0972764
4  Acknowledgement recieved for chunk 2: 1663488044.2026844
5  Time of sent for chunk 4: 1663488044.264632
6  Acknowledgement recieved for chunk 4: 1663488046.4553761
7  Time of sent for chunk 5: 1663488046.6358628
8  Acknowledgement recieved for chunk 5: 1663488048.693251
9  Time of sent for chunk 6: 1663488048.7241707
10 Acknowledgement recieved for chunk 6: 1663488050.815119
11 Time of sent for chunk 7: 1663488050.8415828
12 Acknowledgement recieved for chunk 7: 1663488052.9280531
13 Time of sent for chunk 9: 1663488052.9591134
14 Acknowledgement recieved for chunk 9: 1663488055.1299498
15 Time of sent for chunk 10: 1663488055.140432
16 Acknowledgement recieved for chunk 10: 1663488057.2527318
```

Fig: RTT of chunk 2 for client 5 is coming around 1.5 seconds



Part 2: Using TCP for control messages and UDP for data transfer

- **Design of the code for PSP network with n clients:**

  In this section all the control messages like broadcasts, acknowledgements are sent over UDP, and the data transfer takes place via TCP. It has a similar design of synchronizing and communicating across the ports and threads via queues and lists.

  (1) **Server distributing files to the clients**:
  - In this part, as server will first try to send the files to client, server has to be run first.
  - To make the chunks of equal size of 1kB (for easy processing), the textfile was divided into chunks of 1000 characters(bytes) each, and the last chunk was appended with characters # to make its size equal to 1000.
  - Ports were alloted to server and clients for data handling.
  - n threads are created where each thread is alloted to a particular port of the server that is dedicated to a particular client. Similarly n clients threads also work simultaenously.
  - In every consequent chunks, server sends the data by appending its client ID to the front (10 bytes) to the clients via UDP. It waits for timeout 0.1s to recieve acknowledgement "ACK" from the client (via TCP), else it resends it.
  - To keep an account of acknowledgements recieved, a list of "ACKS" is maintained, where each element of the list is alloted to a port. This ack is recieved from another socket of the server side, which on recieving "ACK" makes the element as "ACK". When this is made the server's port proceeds sending the next chunk till all the chunks are send, after which the threads merge.
  - On the client side, UDP socket is formed for each client to recieve requests from server for connections. The client on recieving data sends an acknowledgement "ACK" to the server.

  (2) **Client requesting missing files from server:**
  - Clients then check their database to find which chunks are missing. For each missing chunk, Clients send a request for the chunk via TCP to an alloted port of the server. *TCP being relaible, we need not handle packet loss here*
  - The server's port on recieving the request appends the request to its pending request queue.

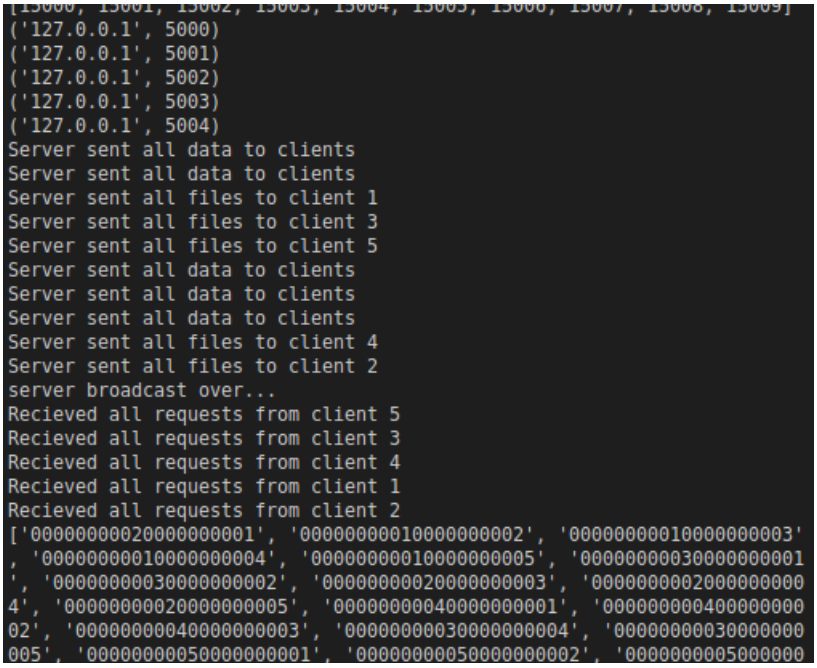
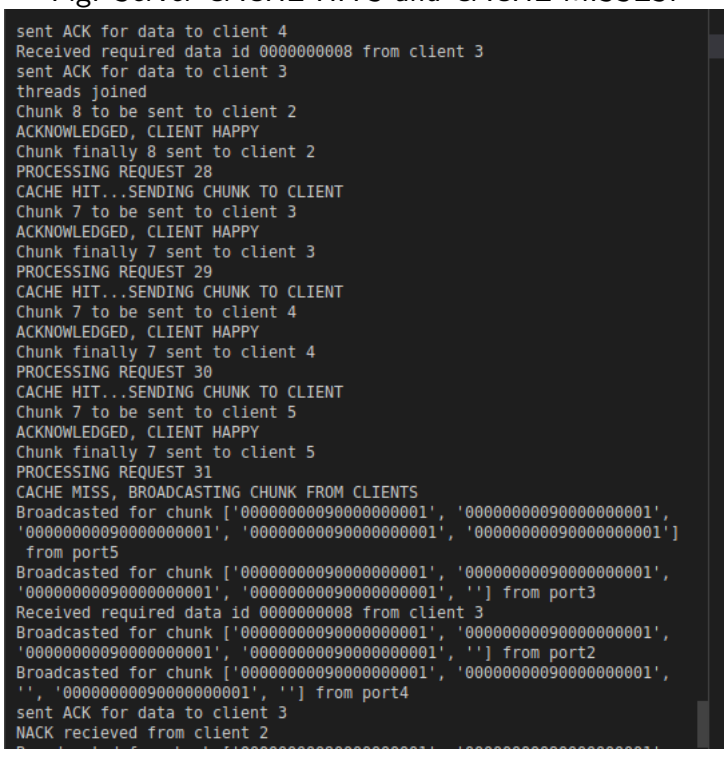


Fig: Server receiving client requests:

(3) **Server checking in the Cache:**

Now, the server checks for the requests in the pending request queue. It pops the request, finds for the chunk in the cache, if present, it sends the data to the client that requires it. Else it broadcasts for the same.

Fig: Server CACHE HITS and CACHE MISSES:



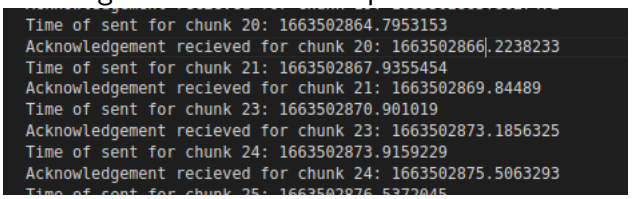
(4) **On Cache Miss, server broadcasts for file from client:**

- Here again threads run in parallel requesting the file for the chunk from each client via TCP. Assuming reliable data transfer via TCP, the requests are sent to client and then code proceeds.

- Client on recieving the broadcast keeps it in a queue. Now the client checks if the data chunk is present in its database. If so, the client sends the data, else it sends "NACK" to the server's port via UDP. All this work in separate threads, so separate elements of a list work as queue for each thread for non-confusion.
  But as UDP is non-relaible, here again it waits for a timeout of 0.1 seconds to recieve acknowledgement via TCP, or else resends.
- The server then recieves the data/NACK via its UDP sockets, and if the data is recieved, then it appends it to the cache and sends it back to the client which requires it.

- **RTT (Round Trip Time):**
  - I have taken wait time *(timeout) of 2 seconds* for the client to send request.
  - Difference in time between sending requests and recieving acknowledgement : 2.464 seconds (on average) But this includes the sleep time (2 seconds), so we need to subtract that time 2.464 seconds.
    So RTT = 2.464-2 = **0.464 seconds** *(on average)*
  - The average RTT for each chunk is coming in the range: 0.43-0.58 seconds
  - The RTT was almost constant for all chunks, except for the first few chunks, for whom it was longer. **As TCP is a reliable service, it ensures packet delivery. So RTT is almost always constant**.

Fig: RTT for client requests via TCP



2. Analysis

:

(1) RTT ANALYSIS:

- The RTT of the first part was slightly less than that of second part.
- In part 1, due to client sending data requests to randomized ports via UDP, packet losses happen sometimes due to congestion which account for peaks in RTT graph at some chunks.
  But in case of sequential(assigned) ports, there was almost no loss of packets or data.
- In part 1, UDP requests will take 3 times the time taken by TCP data transfer. (due to acknowledgement thing in UDP to avoid packet loss)
- In part 2, UDP data will take 3 times the time taken by TCP requests. (due to acknowledgement thing in UDP to avoid packet loss)

- So the total RTT in both the parts remain nearly same. But if there is packet loss in part 1, due to randomization of ports and congestion, there will be an addition of UDP time in the RTT, which causes hike.

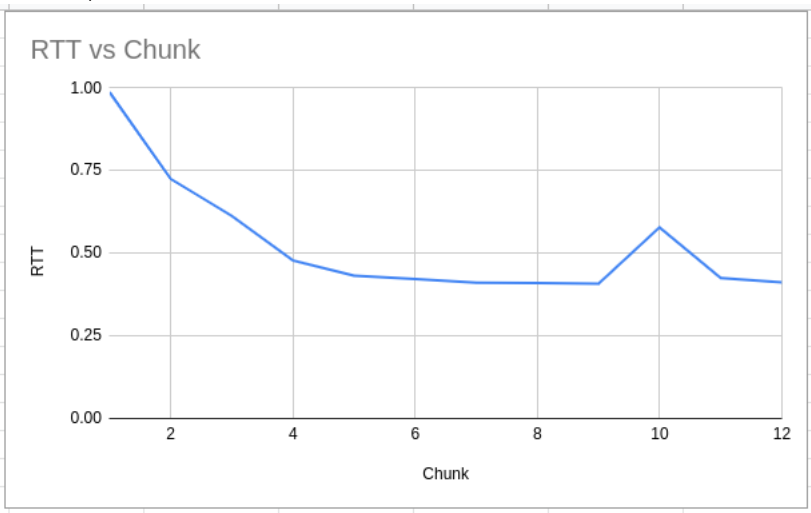


Fig: RTT vs CHUNK for part 1, n=5

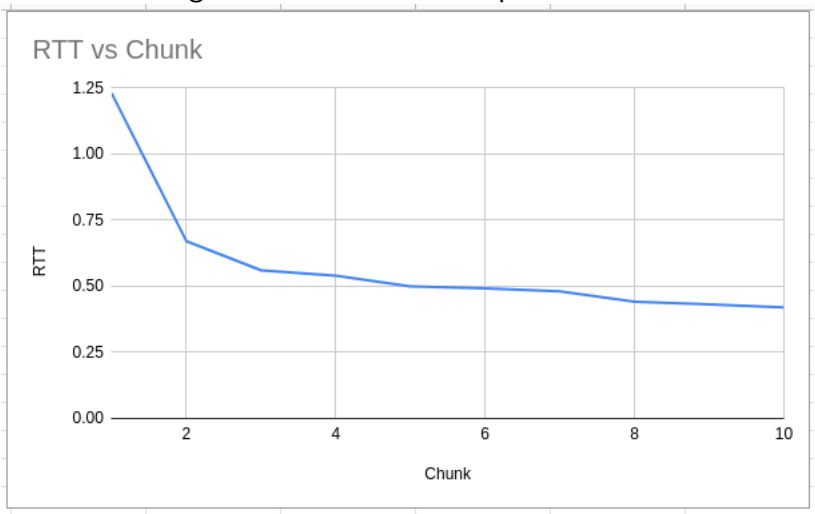


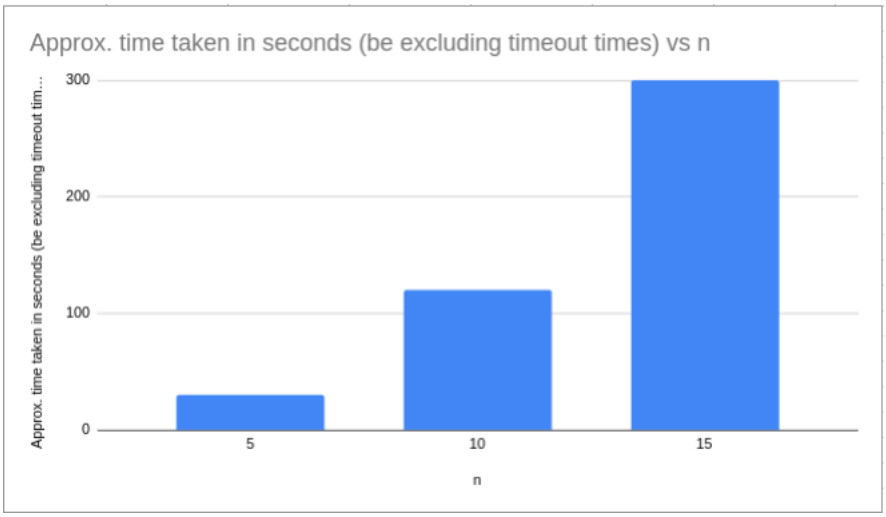
Fig: RTT vs CHUNK for part 2, n=5

(2) **Scalability:**

For n=5, the total time taken was around 8 minutes, but this includes several timeouts as well. So the total time can be said to be around 30 seconds.

n=10, 120 seconds

n=15, 300 seconds



(3) **Cache size and Large file:** For large file, the code didn't run, it stopped unexpectedly. And for large cache sizes and large clients n=100, the code could not run

(4) **Randomized vs Sequential:**

Due to randomization of ports and congestion, there will be an addition of UDP time in the RTT as packets will be dropped and resent, which causes hike in time taken.

For part 1, total time taken for n=5 was around 8 minutes. (randomized)

For part 2, total time taken for n=5 was around 6 minutes. (sequential)

3. Food for Throught

:

- If we see the scenario in real life, it would not be very helpful. The crucks behind this is for every chunk needed if that chunk is already present with server it would take $t_{req} + t_{data}$ but now its taking more time. Even if there is cache hit, real life probability for cache hit on the basis of LRU policy will be very low.
- As the server is mediator, if some clients are inactive, we dont have to worry.
- Every chunk of every file will need two addresses, that is two fields in header one is file id and one is chunk id.