Assignment 2 (Computer Networks)

Vishwas Kalani - 2020CS10411

September 2022

1 Details of submission

- Submission folder contains 8 files :

1. 2020CS10411_server.py
2. 2020CS10411_client.py
3. 2020CS10411.pdf
4. 2020CS10411.sh (shell script for part 1, $n = 5$)
5. 2020CS10411_server_part2.py
6. 2020CS10411_client_part2.py
7. A2_large_file.txt
8. A2_small_file.txt

- The number of clients is a represented by variable **noc** on the top of both server and client files. It can be changed for testing purpose.
- The name of file to be distributed to clients can be given on the top of the server file to the variable **inputFile**.
- The MD5 sum of all the files is printed in the terminal. The files received by client are created in the same folder and along with those files a *logfile* of both server and client is created displaying the exchange of packets.
- I faced an issue in my system that consecutive runs without any delay some times caused socket in use error (even after properly closing the sockets) because the socket accepts and uses next connection after some delay so please it is **best to keep some time gap for multiple simulations**.

2 Details of program

2.1 Flow of program

Part 1 :

1. n threads of the server send initial chunks of each client to it using a *TCP* connection. They send the *id* of the chunk and the total number of chunks in the file to be transferred to the clients as well.
2. Each of the client looks the chunks which are missing in the file, and request the threads of the server for missing chunks using *UDP* connection. The client will send their next request only after they have received the chunk from the server.
3. The server looks into the cache for the chunk requested and if found, sends through a *TCP* connection. Otherwise it sends a broadcast to all the clients one by one asking using *TCP* connection. If it finds the packet before sending to all the clients, it stops and doesn't send request to other clients.
4. The server received the chunk from the client and sends it to the client which requested it.
5. Each chunk received by client is stored in a dictionary.
6. Functions like accessing the cache and broadcasting are locked so that multiple threads don't access them at the same time.
7. Finally all the chunks of each client are merged at correct positions and then written into a file. And then *MD5 hash* function is applied on that file.

Part 2 :

1. The flow of the program remains the same except the fact that all the data transfer including initial chunk distribution, getting chunk from cache or after broadcast is done using UDP.

2. **Handling packet loss :** UDP packet loss has been handled using the socket timeout feature. For a connection, a timeout is kept on both the ends of the connection. If server receives the packet from client before timeout, it sends a positive acknowledgement and the communication continues usually. If timeout occurs before receiving the packet, it sends a negative acknowledgment. If the client receives negative acknowledgement, it sends the packet again. If the acknowledgement is lost and a timeout occurs at the client side, it still sends the same packet again.

2.2 Socket usage

1. **Part 1 :** $3n$ TCP sockets and n UDP sockets have been used.

   - n TCP sockets for initial chunk distribution.
   - n UDP sockets for requesting chunks.
   - n TCP sockets for broadcasting.
   - n TCP sockets for data exchange during broadcasting.

   n threads were active in the server file at any point of time where as n threads were active for the client file during initial chunk distribution and $2n$ threads were active during broadcasting and requesting chunks simultaneously.

2. **Part 2 :** $2n$ TCP sockets and $3n$ UDP sockets have been used.

   - n UDP sockets for initial chunk distribution.
   - n TCP sockets for requesting chunks.
   - n TCP sockets for broadcasting.
   - n UDP sockets for data exchange during broadcasting.
   - n UDP sockets for data exchange from cache.

   n threads were active in the server file at any point of time where as n threads were active for the client file during initial chunk distribution and $2n$ threads were active during broadcasting and requesting chunks simultaneously.

3 Design and implementation decisions

- The LRU cache has been implemented using ordered dictionary data structure of python.
- All the connections in the server and client file have been opened at the beginning of the simulation and closed after the threads of data transfer are complete.
- The broadcast for chunks not found in cache has been made sequentially. Every time a thread of server wants to broadcast, it would broadcast to client 0 in first position.
- Along with the initial chunks distributed to each client, each client also receives the total number of packets present in the file and the number of packets it is going to receive. This information helps in termination of requests from client side.

4 Analysis

1. **Analysis for small file with 5 clients :**

   (a) *Total average RTT part 1* $= 0.0004224915956628733 5s$

   (b) *Total average RTT part 2* $= 0.0009620456859983247s$

   RTT is higher in case of part 2 although it was expected that RTT would be lower in part 2 since most of the data transfer in its case was done by faster transport protocol UDP but since acknowledgements and timeouts have been used in part 2 for handling packets loss, the data of part 1 comes out to be smaller.

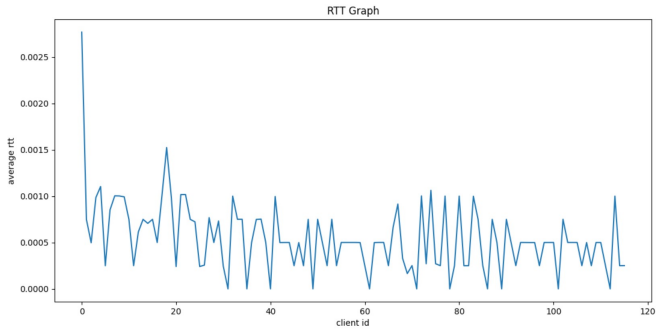

Figure 1: Part 1

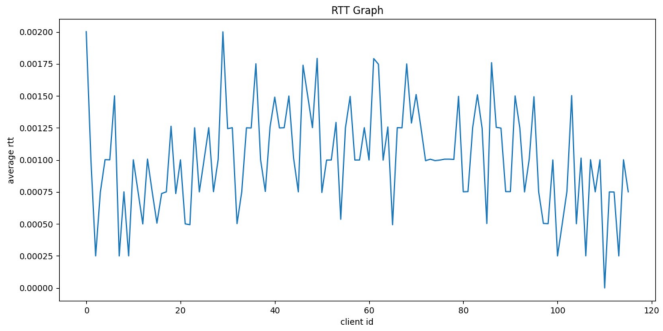

Figure 2: Part 2

2. The RTT for each chunk across all the clients for both the parts has been plotted below :
   The RTT for most of the chunks in both the parts are nearly same but we can see that for some chunks, RTT dips down to very low values as these chunks might have been picked from the cache whereas some chunks are taking longer as they are coming from other clients after broadcast.

3. Total time of simulation in seconds vs the number of clients in seconds is shown in the table and plot below :

Clients	5	10	50	100
Part1	0.2647	0.7218	1.3839	2.9964
Part2	0.5401	0.6052	2.1349	4.1547

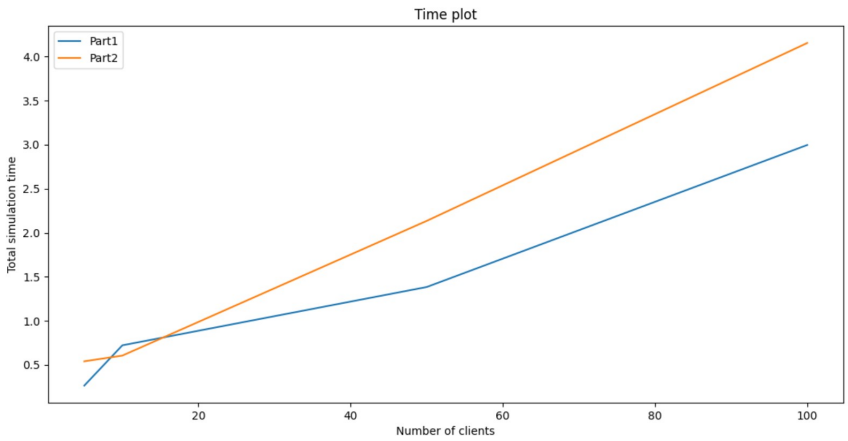

Figure 3: Total simulation time

The plot shows linear variation of simulation time with the number of clients. This behaviour was expected because the number of sockets and the number of threads are both linearly dependent on n, the number of clients.

4. The simulation was very time consuming for the large file with 100 clients because of the limitations of the system to keep large number of threads open for such long time. I tried with large cache size 500 which took ≈ 150 minutes to simulate and cache size 1000 which took ≈ 90 minutes to simulate. This is expected behaviour as the time will reduce on increasing the cache size.

5. When the requests are sent sequentially it takes $0.26s$ where as if the request are sent in random order $0.2853s$. I used the random shuffle function of python to randomly shuffle the list of packets at each client. ($n = 5$, Part 1). This is expected behaviour since in sequential request, there would be multiple occasions that the clients would request a packet around same time and thus it could be accessed from the cache.

5 Food for thought

1. There are various advantages of PSP network over traditional file distribution for instance the data is being sent in chunks which is reducing the load of both server and the connection sockets. The server acts as a mediator for the clients to transfer the file and this avoids large file storage on server.

2. In traditional P2P networks unlike P2S networks, there is no central authority which can authorize and permit the transmission of legally allowed data. The server can also ensure the authenticity of data transmitted between the clients in case of P2S network.

3. If there are multiple files and each clients wants one of those, then the chunk distribution doesn't change and it can be either sequential or random. We can append the file id ahead of every chunk keeping let us say initial 3 digits fixed for it. In this way the client can check if they need a particular chunk or not.