File Structure

The following files are present in the folder submitted:

1. $LRU.py$: Contains the implementation of the Least Recently Used Cache.
2. 2020CS10336_ < client|server > _part < 1|2 >.py: Contains the client and server files for part 1 and for part 2.
3. 2020CS10336.sh: Contains the shell file to run the simulation for part-1 with 5 clients and small file.
4. The programs generate a log file with RTT values. They will be names as part-1-RTT.log and part-2-RTT.log.
5. The client files have commented code at the end to generate a CSV file with the average RTT values for each chunkID across all clients. These CSV files will be named part-1-RTT.csv and part-2-RTT.csv. (Delete these files if you run the commented code)
6. The two text files used in the assignment are A2_small_file.txt and A2_large_file.txt
7. The program will also generate txt files at the end of the simulation for each client with names 1.txt, 2.txt . . .

Note: The arguments for the simulation run are within the client and server files. For e.g the number of clients, the txt file to be used, cache size etc. are specified as global variables in the client and the server file. I have set them to default values of 5 clients, small file, sequential requests and cache size = number of clients The shell file runs Part-1 for these values.

```
$ ./2020CS10336.sh
 Simulation took 1.3645586967468262 seconds
hash of the input file is 9f9d1c257fe1733f6095a8336372616e
Time Take to receive all chunks by client 4 is 1.0051658153533936 s
Time Take to receive all chunks by client 1 is 1.006155252456665 s
Time Take to receive all chunks by client 5 is 1.0041539669036865 s
Time Take to receive all chunks by client 3 is 1.0051658153533936 s
Time Take to receive all chunks by client 2 is 1.0051658153533936 s
Client 1 took 1.1816086769104004 s to recieve all packets
Client 4 took 1.2533116340637207 s to recieve all packets
Client 5 took 1.252528190612793 s to recieve all packets
Client 2 took 1.2687108516693115 s to recieve all packets
Client 3 took 1.2687108516693115 s to recieve all packets
Client Number 3 is closing UDP socket to listen for broadcasts
Client Number 2 is closing UDP socket to listen for broadcasts
Client Number 4 is closing UDP socket to listen for broadcasts
Client Number 1 is closing UDP socket to listen for broadcasts
Client Number 5 is closing UDP socket to listen for broadcasts
Client number 3 has file with hash 9f9d1c257fe1733f6095a8336372616e
Client number 4 has file with hash 9f9d1c257fe1733f6095a8336372616e
Client number 5 has file with hash 9f9d1c257fe1733f6095a8336372616e
Client number 2 has file with hash 9f9d1c257fe1733f6095a8336372616e
Client number 1 has file with hash 9f9d1c257fe1733f6095a8336372616e
```

Running the sh file

Part 1

Architecture explanation

There is a fixed TCP port on the client looking for clients to join the PSP network (designated as *TCPServerSocket* in the server side code). Each client is implemented as a thread in the client side code. Initially each client sends requests to the *TCPServerSocket*, the server accepts these requests and spawns several threads to handle each of these clients.
One of these threads is for the initial transfer of data to each client. The server first divides the file into chunks

of size 1Kb and tells each client the number of total chunks in the file. Then the server divides the chunks among the clients in a round-robin fashion. If the number of clients is n then the first client receives packet number $[1, n+1, 2n+1 \dots]$, the second client receives packets $[2, n+2, 2n+2 \dots]$ and so on. This ensures every client receives about the same number of chunks. The server also sends a termination signal to all the clients to indicate that the initial transfer is over.
The clients store the chunks they have received in a dictionary and they now know which packets they don't have and have to request from the server. Depending on the request scheme each client can request the missing packets in a sequential or random fashion.
The server spawns *n* threads each of which has a *UDP* socket listening for the client requests. Each client also knows the port of the server *UDP* socket listening for the requests from that particular client.
The client has a UDP port which it uses for requesting chunks to the server (reuses the same UDP port for multiple requests). The client looks for a missing packet and requests that packet to the server using this UDP port. As described before the server has a thread to handle these UDP requests from the client.
The server first checks if it has the requested packet in the cache and if not it sends out a broadcast to each of the clients (each thread of the server listening for requests from a different client and has one port to send broadcasts which it reuses). To do this the server uses a UDP port to request the chunk to all the clients.
Each client also has a UDP port to listen for broadcast requests from the server and each client spawns a thread to listen for broadcast requests from the server.
The server also has n threads to listen to broadcast responses from each client. Once the server receives the responses to a broadcast it puts in the cache and also sends it to the client that requested it.
Note: I have tried to keep all connections as persistent as possible i.e if a socket can be re-used I reuse it without closing it. This saves time required in making initial TCP connections or binding ports.

Server Side ports

TCP Ports

1. 1 TCP port to listen to all initial requests from clients. Each of this leads to n connection TCP ports with each client.
2. n to listen to broadcast request responses from each clients. (these are threaded and can happen concurrently).
3. n to reply to the client requests (these are also threaded), defined in the threads listening for chunk requests, each such thread has a TCP connection with the client to send requested packets.

Thus there are $3n + 1$ TCP ports on the server.
UDP Ports

1. n UDP ports to listen to client requests concurrently.
2. n UDP ports to send broadcast requests to the clients.

Thus there are total $2n$ UDP ports on the server side.

Client side ports

TCP ports

1. 1 socket per client involved in the initial transfer of chunks.
2. 1 socket per client to reply to broadcast requests.
3. 1 socket per client to listen for replies to chunk requests to the server

Thus there are *3n* TCP ports across all clients on the client side.
UDP ports

1. 1 socket per client to listen to broadcast requests from the server.
2. 1 socket per client to requests missing chunks to the server.

Thus there are total *2n* UDP ports across all clients on the client side.

Part 2

Architecture explained

It is very similar to part-1. Part-2 still uses the initial TCP socket to establish the connections between the clients and the servers.
The initial packet exchange now happens through UDP. I use acknowledgments to handle packet drops.
The client now uses a persistent TCP socket to request for missing chunks. The server has n threads with client-specific TCP ports listening for packet requests.
Each thread of the server listens for chunk requests and if the server does not have the requested packet in the cache it has to send TCP broadcast messages to every client. To do this it creates and a TCP socket (without binding) for each client sends a request and closes it. Note that these connections are not persistent.
Every Client has spawns a thread with a TCP socket listening for broadcast requests. If this client has the packet it responds to the request using a persistent UDP connection.
There are n threads on the server side listening for broadcast request responses for each of the clients. Again UDP packet drops for these exchanges are handled through acknowledgments.

Server Side Ports

TCP ports

1. 1 initial TCP port listening for initial client connections. This will further create n TCP connections with each of the clients involved in the initial transfer.
2. n TCP sockets listening for client requests for missing chunks
3. Each thread listening for requests can create at most 1 TCP at a time to send broadcast (note that these connections are not persistent) Thus there are $3n + 1$ TCP ports on the server side.

UDP ports

1. 1 initial UDP port per client for initial transfer.
2. One persistent UDP per client to respond to chunk requests
3. One UDP port per client to listen for broadcast responses from clients.

Thus there are total $3n$ UDP ports on the server side.

Client Side Ports

TCP Ports

1. One TCP connection per client for the initial transfer
2. One persistent TCP connection to request missing packets
3. One persistent TCP connection to listen for broadcast requests

Thus there are total $3n$ TCP sockets on the client side for all clients combined.
UDP ports

1. One socket per used for the initial transfer. This socket is re-used to receive responses from the server
2. One socket per client to reply to broadcast requests (kept persistent)

Thus there are $2n$ UDP ports on the client side.

Handling UDP packet drops

For part-1, I used an indirect way to handle UDP packet drops, since in part-1 UDP messages are control messages. I resend UDP control messages whenever the intended effect of the control message does not take place after a specific amount of time.
For eg. If a client requests a missing chunk using UDP message and for it doesn't receive the chunk in the next 1 s (due to packet drop or other reasons), I re-request the chunk by resending the UDP control message.
Similarly, If the server sends a broadcast using a UDP message and it doesn't receive the chunk from the clients in 1 s (due to packet drop or other reasons), it resends the broadcast.

For part-2, UDP packets contain file data, to handle packet drops in this case I use acknowledgments and timeouts. After sending any data packet through UDP, I wait for an acknowledgment response. In case the timer times out and I don't receive the acknowledgment (either because of packet drop of the initial message, its acknowledgment or some other reason). I resend the packet.

Analysis

1. The client side generates a log file for both parts which have the average RTT values of each client across all chunks. The results are as follows

```
2022-09-20 16:22:42,648 Average RTT for all requested packets for Client 1 is 0.0069257202355758
2022-09-20 16:22:42,648 Average RTT for all requested packets for Client 3 is 0.00384193851101783
2022-09-20 16:22:42,649 Average RTT for all requested packets for Client 4 is 0.0038383058322373257
2022-09-20 16:22:42,650 Average RTT for all requested packets for Client 2 is 0.0039060782360774214
2022-09-20 16:22:42,650 Average RTT for all requested packets for Client 5 is 0.0037943419589791245
```

Part-1

```
2022-09-20 15:53:41,536 Average RTT for all requested packets for Client 2 is 0.017327916237615 77
2022-09-20 15:53:41,544 Average RTT for all requested packets for Client 3 is 0.017343767227665 07
2022-09-20 15:53:41,544 Average RTT for all requested packets for Client 1 is 0.017859236053798 51
2022-09-20 15:53:41,557 Average RTT for all requested packets for Client 4 is 0.018150978190924 53
2022-09-20 15:53:41,564 Average RTT for all requested packets for Client 5 is 0.017990978815222 3
```

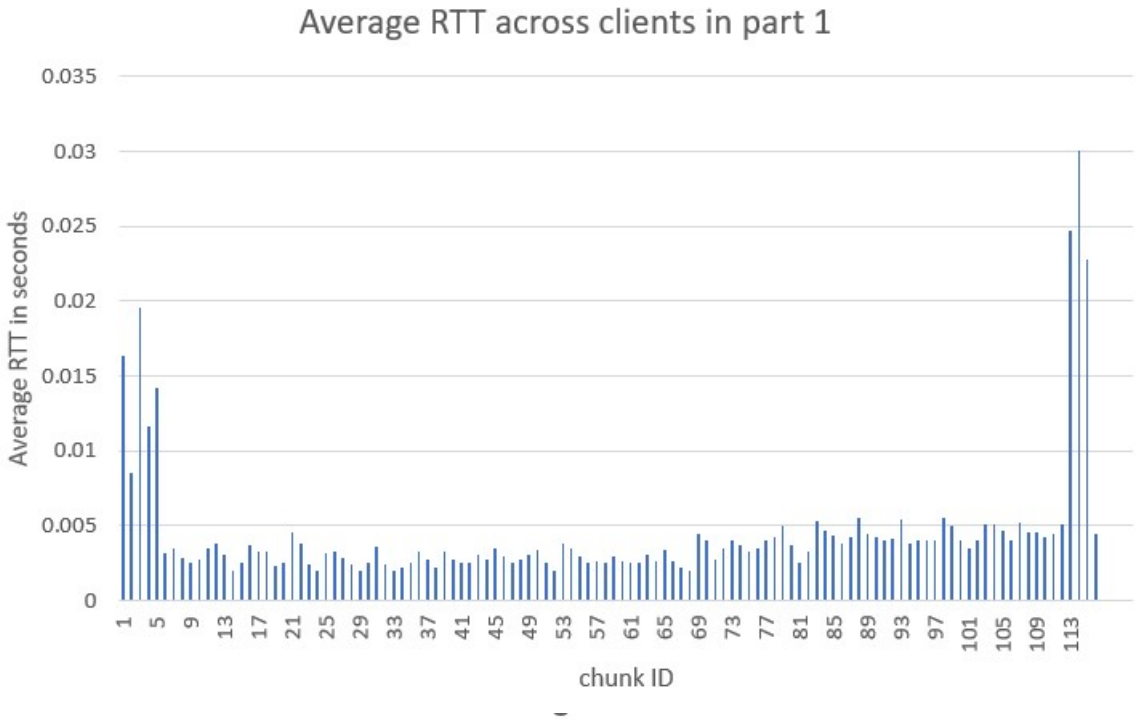
Part-2

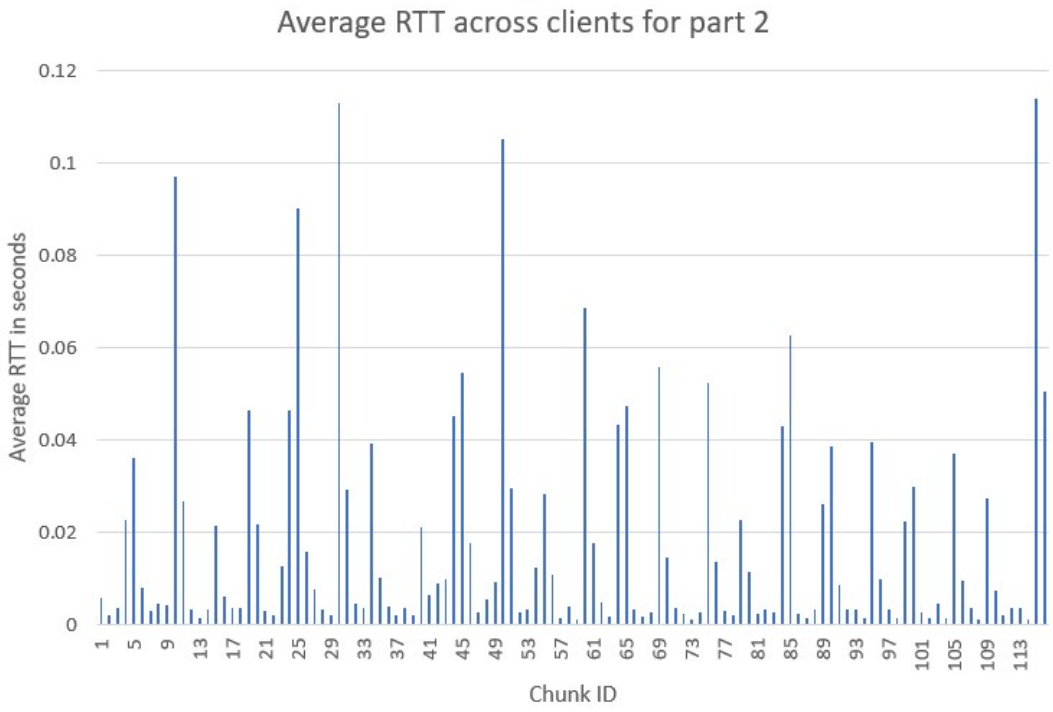
All though UDP packet exchanges are expected to be faster. In this case TCP simulation ran faster this is primarily because of two reasons.

(a) We have handled UDP packet drops explicitly using acknowledgments and timeout. This is expected to increase the simulation time

(b) All connections in simulations of part-1 were persistent whereas in part-2 the TCP broadcasts requests from the server side were not persistent, every time we make a broadcast request we establish a new connection. This is expected to increase the simulation time.

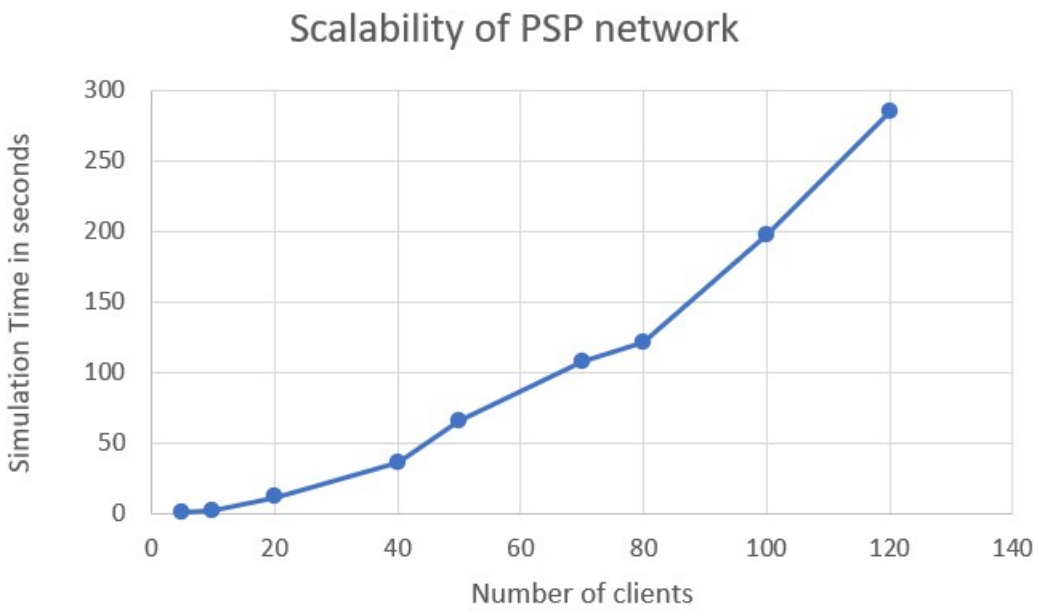
2. For both parts there is code (commented) in the client side to generate a CSV with average RTT values for each chunkID across all clients. The results are as follows





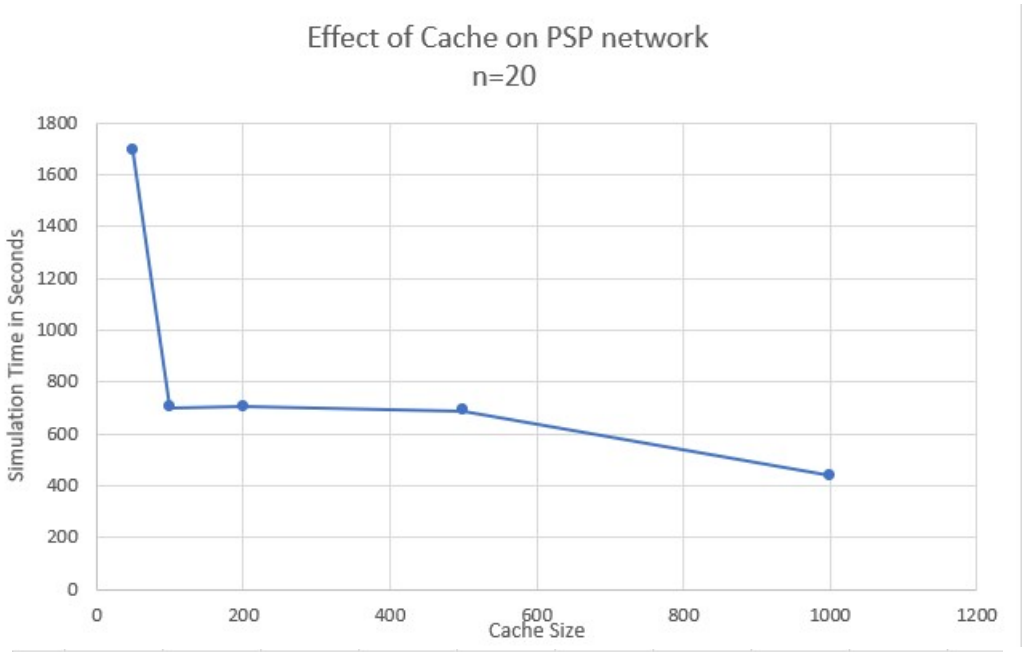
There are some packets with a higher RTT compared to other packets. This might because of packet drop where some packets are re-requested either from client side or server side. Or it might be that they are not in the cache and the server has to send broadcast more than once to get them (i.e it was in the cache before but got evicted and the server has to resend a broadcast it had sent earlier to get this packet).

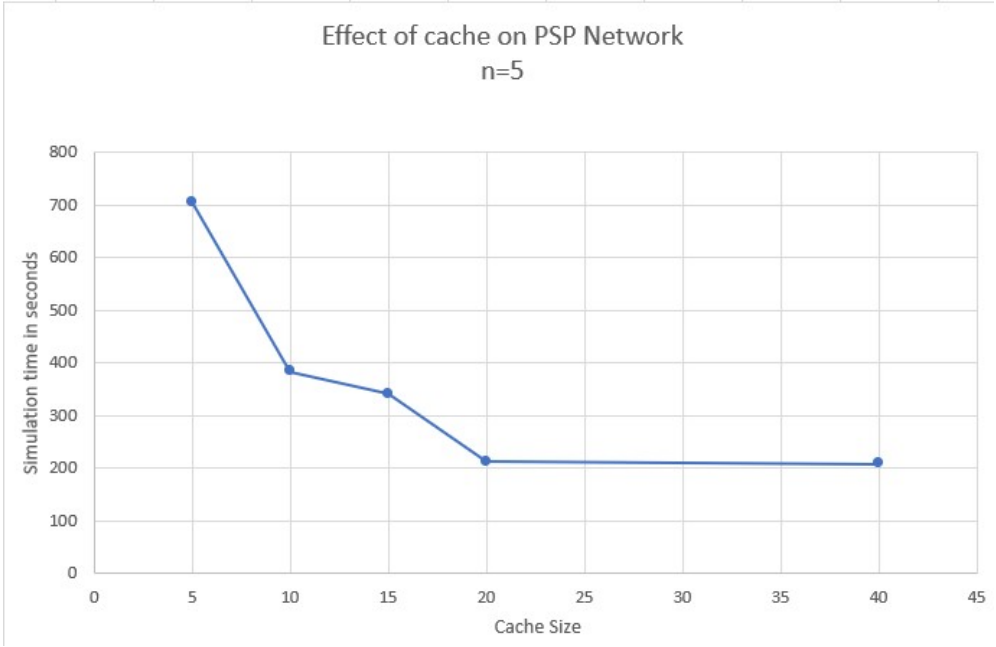
3. I ran the simulation for the small file with varying number of clients for part-1 with cache size equal to the number of clients.



PSP network is clearly not scalable compared to the P2P network, since all requests are directed through the server. For small number of clients, the server can reply to all the clients concurrently. But as the number of clients keeps increasing it becomes increasing difficult for the server to reply to all clients concurrently. Also the server sends out to broadcast to many clients for a packet, if many clients have that packet than many clients will return the same packet to the server which is redundant and causes server congestion. There is an exponential rise in the time taken as the number of clients keeps increasing while it is somewhat constant initially.

4. For this I use the large file and run part 1, I keep the number of clients to 20 (because of system constraints) and keep increasing cache size and note the simulation time. I also do it with 5 number of clients.





As the cache size increases it is more likely for requested chunks to be in the cache preventing the need for broadcast requests. With smaller cache sizes the server has to send multiple broadcast requests since packets might be evicted from the cache. The gain in time decreases as we approach the case where the server always has the chunk in cache without having to broadcast more than once for the same packet.

5. I ran the simulation for part-1 with 5 clients and the small file, the cache size was kept as 5 i.e. the number of clients. In sequential requests the clients request chunks which they don't have in a sorted order, on the other hand in the random case every time the client chooses a random chunk to request out of the chunks that it doesn't have.

Random and Sequential Requests	
Random Requests (s)	Sequential Requests (s)
2.57	1.26
2.63	1.29
2.04	1.27
2.49	1.3
1.82	1.25
Average: 2.31 s	Average: 1.274 s

This happens because of the caching policies we have used. All clients are running on separate threads and requests chunks to server simultaneously. In case of sequential requests many clients requests the same chunk to the server simultaneously and it is very likely for that chunk to be in the cache. For eg. 4 clients will request for chunk 1 to the server. Once the server receives this chunk it will simultaneously send it to all four of them. On the other hand if packets are requests randomly then it is less likely for that packet to be in cache and the server will have to keep sending broadcasts to re-request for this chunk. Because of the small cache size, that packet will be eventually removed from the cache and the server might need to re-request it if some other client requests it.

Food For Thought

1. The server can save space in this model by deleting the file after sharing it once and maintain a much smaller cache, this helps save memory on the server.
2. P2P networks face the problem of security as it is difficult to ensure the authenticity of the data being shared among the peers, in this architecture the server can act as an intermediary to ensure authenticity of the data being transferred. Also every peer has to use extra-compute because they have to maintain connections with several peers. Every peer also has to maintain other statistics about the peers it is connected to. In the PSP architecture every peer just has to maintain information about the server and has only one connection with the server.
3. Currently every chunk has 1024 bytes of data and a header appended at the end which is used to denote the chunkID. If there are multiple kinds of files then we will also have to store information about which file that chunks belongs to in the header. We can append some bytes which stores the fileID along with the chunkID. Apart from this change other parts of the simulation remian mostly similar, while requesting a chunk the client also has to indicate the fileID of the requested chunk along with the chunkID. Whenever the client or server receives a chunk it has to check its fileID along with its chunkID.