COL334 Assignment 2

Rishi Jain
2020CS10373

September 2022

Note:- Shell script file uses gnome command for opening terminal which runs in ubuntu machines only. Please use ubuntu machine for simulation.

1 Introduction

Assigning of ports to server and client is as follows:-

For Client: -
Each client has 2 access to two ports. One for receiving chunks from the client and other for receiving requests for chunks from the server. For sending request or a chunk to the server, Client creates a socket without binding and sends data over that socket.

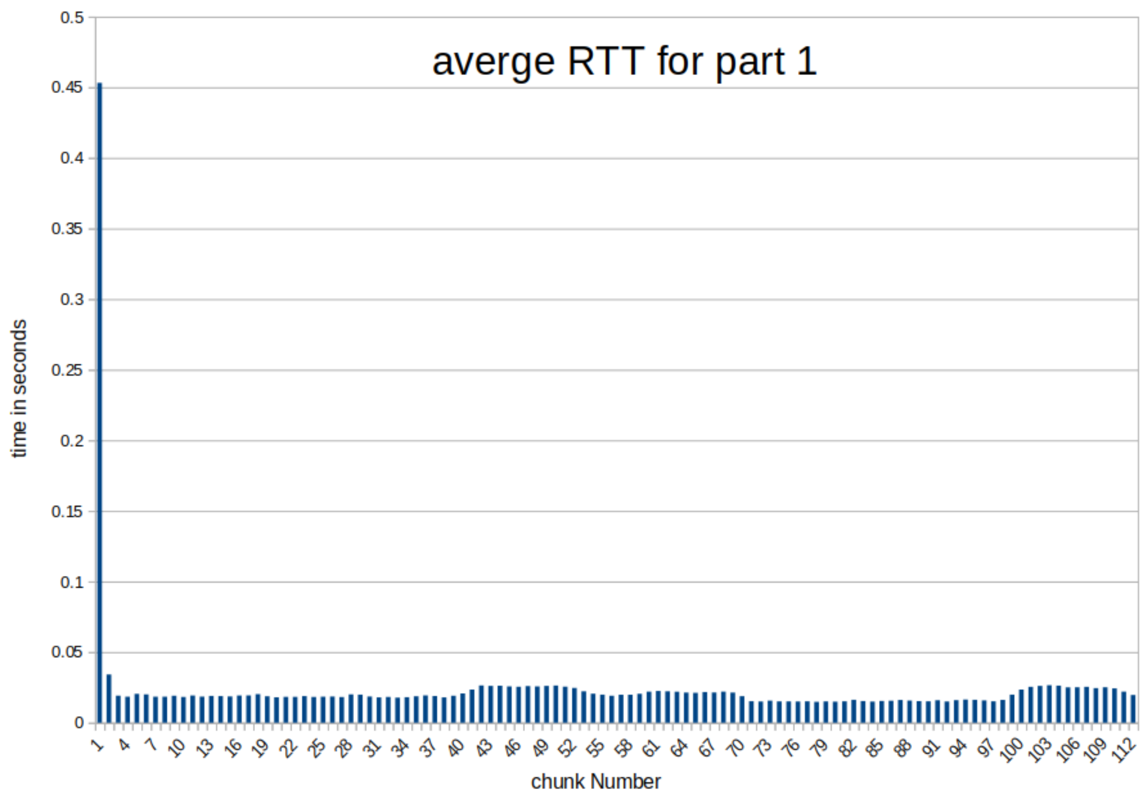
For Server: -
Server has access to 2*n ports. For receiving chunk request, I've assigned n ports and n for receiving request from the client. For sending request or a chunk to a client, Server creates a socket without binding and sends data over that socket.

UDP packet drops are handled by sending data again if acknowledgment is not received after 1 second of sending the request. Acknowledgment is also sent using the UDP protocol. If acknowledgment gets dropped sender will send the data again, and thus acknowledgment will also be sent again. This cycle continues till acknowledgment is received by the server.

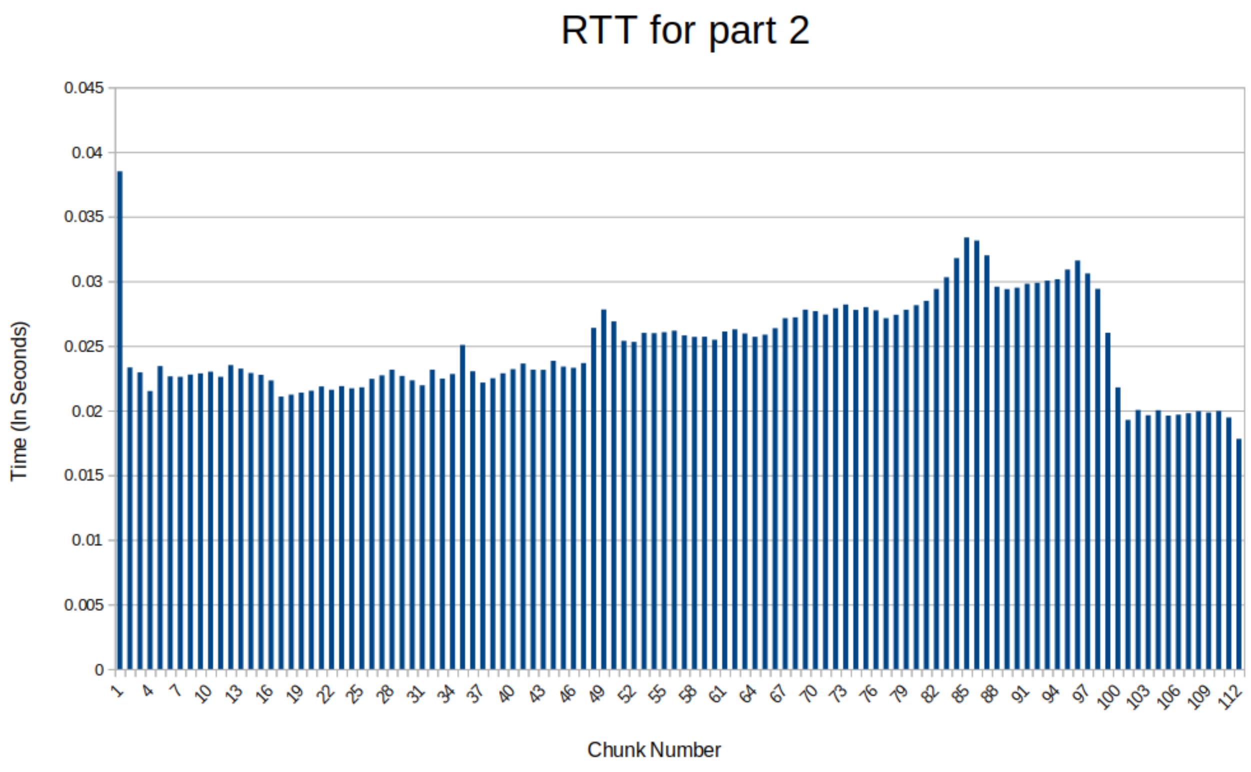
2 Analysis

1) For this part I've used small text file with number of clients(n) = 100 with sequential chunk requests by the client.
The file containing RTT is uploaded as RTT-part1.csv and RTT-part2.csv . For part 1 average RTT is 0.0165706415799101 seconds and for part 2 average RTT is 0.0249659585918355. Clearly part 1 has lower RTT and this is due to the difference in the implementation of TCP and UDP networks. TCP is more organized and takes more time. Whereas UDP doesn't ensures delivery but it is faster than TCP. Since in part 1 request for chunk was sent over UDP, RTT for chunk took less amount of time.

2) The Chunk No vs average RTT graph for part 1 is: -



The Chunk No vs average RTT graph for part 2 is: -



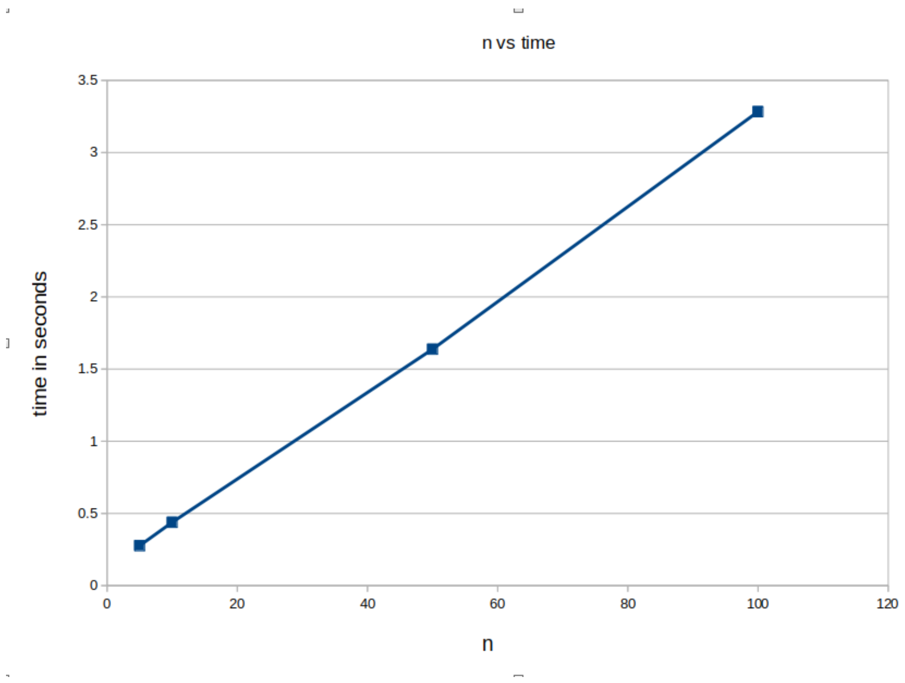
The last column of the RTT-part1.csv and RTT-part2.csv contains the average RTT time for the chunks. The first chunk RTT is significantly higher than the others as initially cache is empty and at the start client whose request is fulfilled start requesting other chunks and might cause eviction of chunk 0 before it gets supplied to every client. Thus chunk 0 has significant high RTT. Since every chunk is requested 99 times, if the cache size is infinite, then we can expect that only the first request for that chunk requires requesting from the clients. Others will be served with less delay. Since the size of the cache is limited, some chunks might get evicted, and thus, their RTT is a little high than others. From the graph, it can be seen that RTT increases in the middle for sequential access. This is due to the reason that in the middle, there are more cache misses. At the start, chunk numbers from a similar range are fetched, but as it goes into the middle, different clients request different chunks, and since cache size is limited, each chunk is fetched many times. In the end, traffic becomes less; thus, eviction of data from the cache is not as fast as in the middle.

3)

a) For small cache size: -

For this part I've fixed cache size to 5 and used small text file for the simulation. The analysis is done for part 1 with sequential chunk requests.

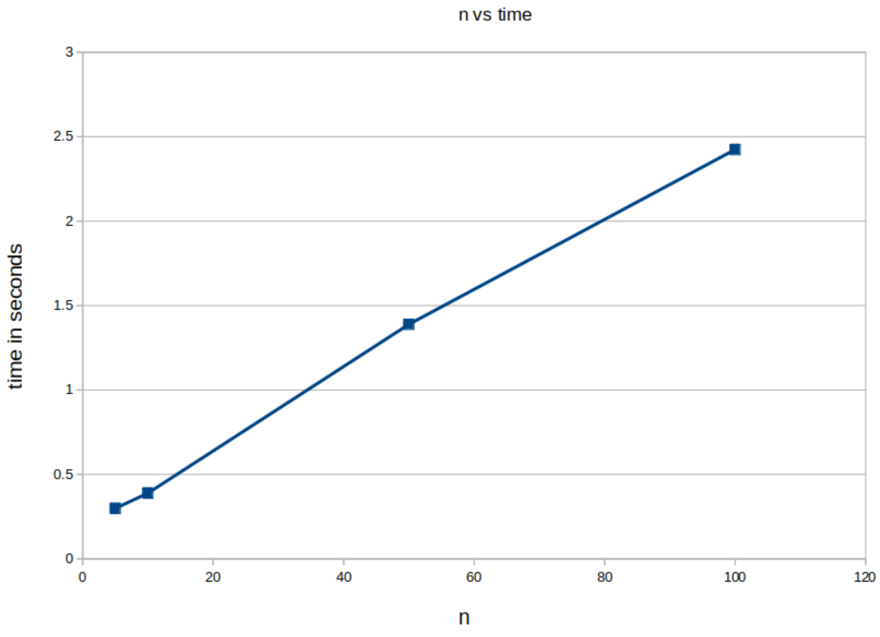
Sr No.	number of clients	time(in seconds)
1	5	0.2765047550201416
2	10	0.43860960006713867
3	50	1.6377675533294678
4	100	3.283201217651367



This Trend was expected due to the limiting cache size and server bandwidth. As n increases, there is more load on the server cache. This frequently leads to the eviction of chunks, and thus, the same chunk has to be fetched several times for different clients. As n increases, the load on the server increases, and thus time also increases.

b) For large cache size: -
For this part I've fixed cache size to 300 and used small text file for simulation.

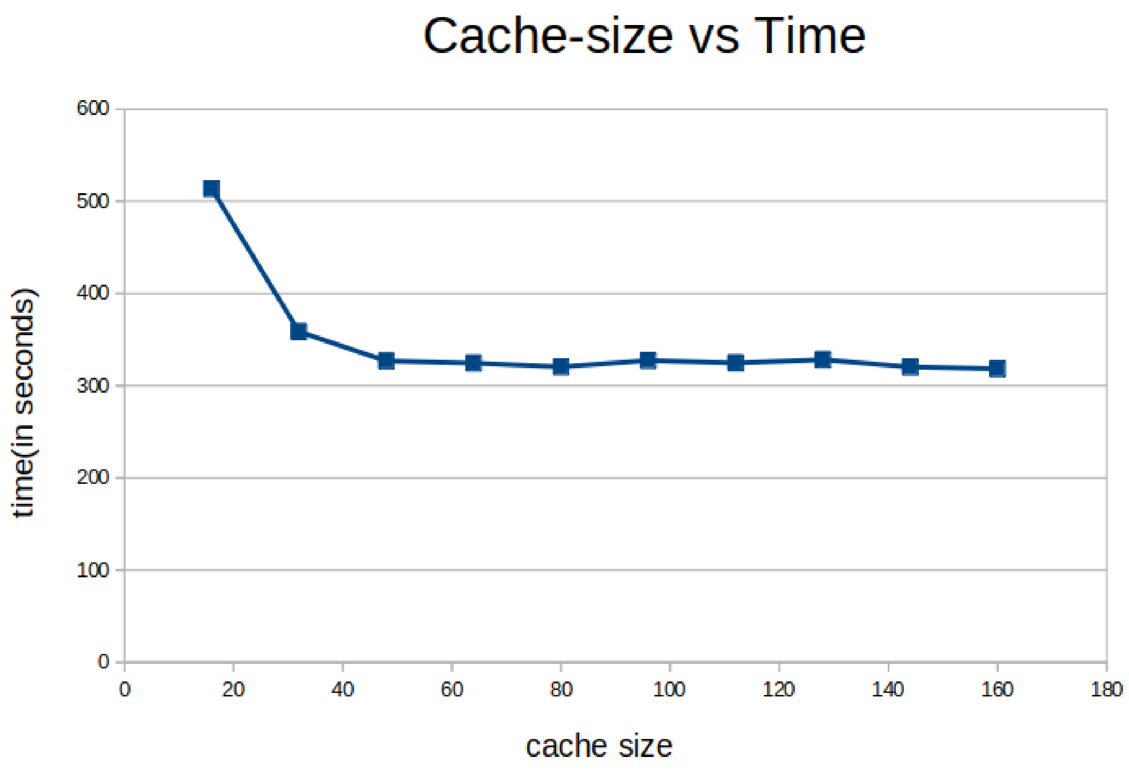
Sr No.	number of clients	time(in seconds)
1	5	0.298849582672119
2	10	0.389237880706787
3	50	1.38897037506104
4	100	2.42405438423157



Since simulation is done in the same computer for a large cache size, n should not matter due to high bandwidth. But since the computer has a limited number of threads, they have to be used repeatedly for various processes, which will lead to delay and thus as n increases delay increases. But we can notice threading by observing the difference when n=5 and n=10, Since these requires small number of threads, they can be simulated in parallel for clients. Therefore, there is only slight difference between RTT for n=5 and n=10.

4) Since the max number of threads in my Computer is 16 and each program creates 2n threads, The value of n for this part I am choosing is 16 . Cache size varies in multiple of 16. Requests are sequential and analysis is done for part 1.

Cache Size	time(in seconds)
8	1306.25312185287
16	475.424424171448
24	299.679035186768
32	298.456642389297
40	301.478289604187
48	313.293785333633
56	300.707428216934
64	298.155016899109
72	308.244018554687
80	299.273038625717



For small values of cache size, the time taken is significantly high; as cache size increases, the time taken by the program becomes constant. For small cache size, many cache miss happens due to different chunk requests by various clients. But since the requests are sequential, all the requests for the same chunk are very close in time(Temporal locality). For tiny cache size even a little variation in the request sequence can be a cause of eviction of essential data. As the cache size increases, due to sequential requests, most of the previous data is not accessed again and remains in the cache for more time. Time delay becomes constant because after some time, data fetched some time earlier becomes useless as it will not be requested again, and thus, it can be evicted from the cache. From the simulation, we can conclude that the optimal cache size should be 48 for n =16.

5) The analysis is done for n=5 with small text file and cache size is fixed to 5.

For requests in random order time taken by the program is 0.835576057434082 seconds and by sequential access time taken is 0.2765047550201416 seconds. Clearly requests in random order takes more time. This is due to the reason that sequential ordering utilizes the temporal locality i.e. request for same chunk are very close in time and thus many chunk requests are served without getting evicted. Whereas in case of random order of requests, many chunks are evicted from the cache before another request for the chunks arrives due to the random order of requests. Thus, random order takes more time than sequential order of requests.

3 Food For Thought

1) One significant advantage of a PSP network over a traditional network is that the server doesn't need to store the whole file and thus reduces memory consumption. Also, since the file is requested in chunks, we can resume the downloading process. Therefore even if the network crashes, we can continue the file transfer and don't lose any progress.

2) Since each connected computer is independent, peer-to-peer networks malware and virus can attack the computers. If one of the computers tends to get virus-infected, it could quickly spread to the remaining computers even if they are protected through antivirus software. In the PSP network, we can check the file for viruses inside the server before sending it to clients. This will reduce the chance of other computers getting infected.

3) When there are multiple files across the clients, and each client wants one of those, we have to identify which file the chunk belongs to for each chunk. Each chunk will have a header that will tell which file the chunk belongs to. We can do this by simply assigning the first 100 characters of the chunk to show which file it belongs to.