Script usage

`python3 client.py ./input.csv`
where ./input.csv can be replaced by path to the input file
The chunk size can be changed in the script itself (currently set to 100k bytes)
The output path is set to ./actual-file-name. To set a custom output file path, set it in the output_path variable, and set keep_file_name_same to False.

Question 1

The script was written in Python3. Steps to download and verify the file were:

1. A socket which can communicate with IPv4 addresses via TCP is instantiated
2. A connection is established with vayu.iitd.ac.in, setting destination port 80
3. The following GET request is sent
   `GET /big.txt HTTP/1.1\r\nHost:  vayu.iitd.ac.in\r\n\r\n`
4. The first 4096 bytes of the response are received. The header is parsed, and it is checked if the response is 200 OK. If so, the content length in bytes is extracted from the header. The content part after this header is extracted. More data is then received till the received bytes of the content (total bytes received - header bytes) are less than the content length. Data is simultaneously written to file in chunks of 4096 bytes.
5. The md5 checksum is calculated and checked.

Question 2

1. The request given in the question is typed in vim in binary mode (special characters typed using ctrl+M). The ncat command was used as given in the question. The text file corresponding to the range was received.
2. Now the python script was edited to include `Connection` and `Range` params of the header. Multiple GET requests were now sent to receive the full file.
3. From the header of the first response, the file size is determined by parsing the `Content-Range` field. The number of chunks are determined
4. For each chunk, GET request is sent, data is received and written to file. In case a response is not received, the socket is closed, and a new socket is opened and connection established, and the request resent.
   At the end md5 is verified.

Question 3

2 thread-safe synchronized data structures were implemented for this part:

1. DataQueue: contains a priority queue to hold chunks, and manages writing them to file.
   Whenever a thread completes a chunk download, it is inserted into the priority queue. Chunks are then written to disk if possible (i.e. if the lowest unwritten chunk is present in the queue, since writing to disk should be in order)

2. TrackChunks: contains the lowest unassigned chunk number, which can be assigned to a thread wanting to pick up a new chunk

A summary of the download process is as follows

1. A GET request for the file with range: 0-0 is sent to one of the servers. The size of the file is parsed from the response from the `Content-Range` field. Total no of chunks are determined using the file size.

2. Threads, equal in number to the total number of connections are started. On each thread, a socket is opened and a TCP connection to one of the servers is established.

3. Each thread does this in a loop: Requests a chunk no. from `TrackChunks`, sends a GET request corresponding to that chunk, and then receives the data (and parses out the header). A new thread is branched to push this downloaded chunk onto the `Dataqueue`, which also handles writing of data to disk.

4. Whenever a new chunk is not available, the socket thread ends. The download is complete when all these threads and the write threads (which branched from them) join.

Effect of varying number of parallel TCP connections

Table 1: Download time from vayu for different no of parallel connections

No of connections	Download time (s)
1	19.06
2	11.15
3	7.38
4	6.67
5	5.38
6	5.28
7	4.01
8	4.88
9	4.61
10	2.98
20	8.2

The download time decreases with more parallel TCP connections, for less no of total connections. This is due to the reason explained in class, as the sum of window sizes summed across all connections generally increases with increase in no of connections (as if the window size falls for one connection, it may still be large for another, so the sum remains large). Thus the no of packets pushed into the network remain large, and the download time decreases.
For very large no of connections (like 20 in case of vayu), the download time increases because so many parallel connections cannot be maintained, hence there are disconnections, so a lot of time goes in detecting timeouts and establishing connections.

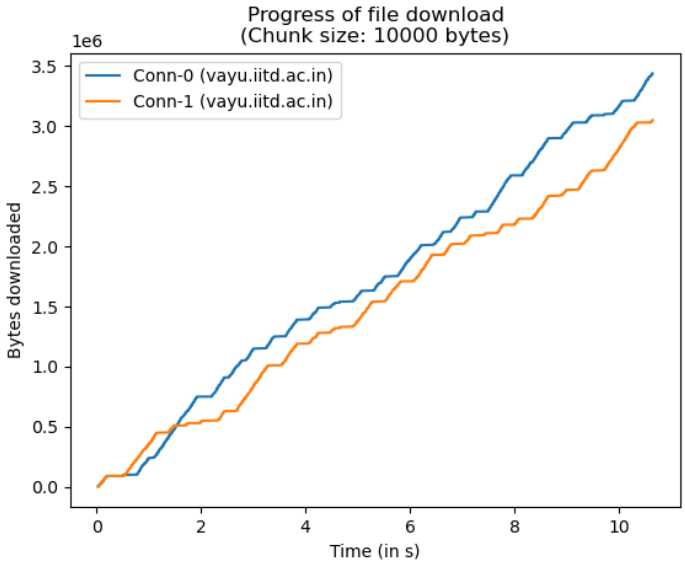

Figure 1: 2 connections

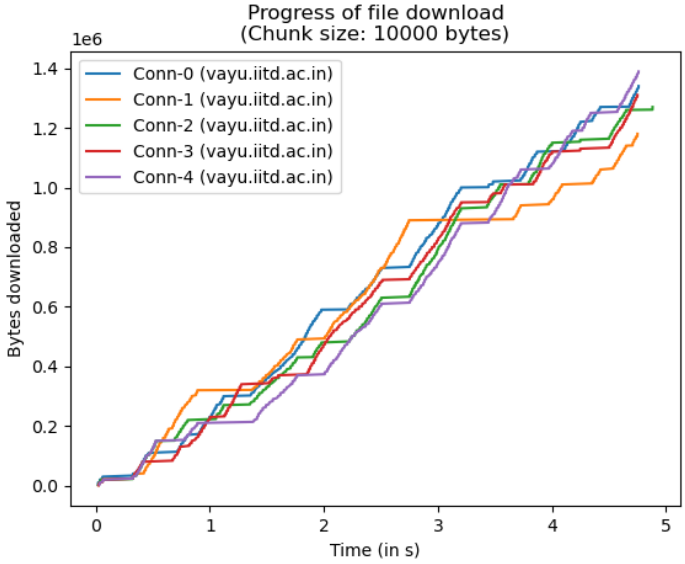

Figure 2: 5 connections

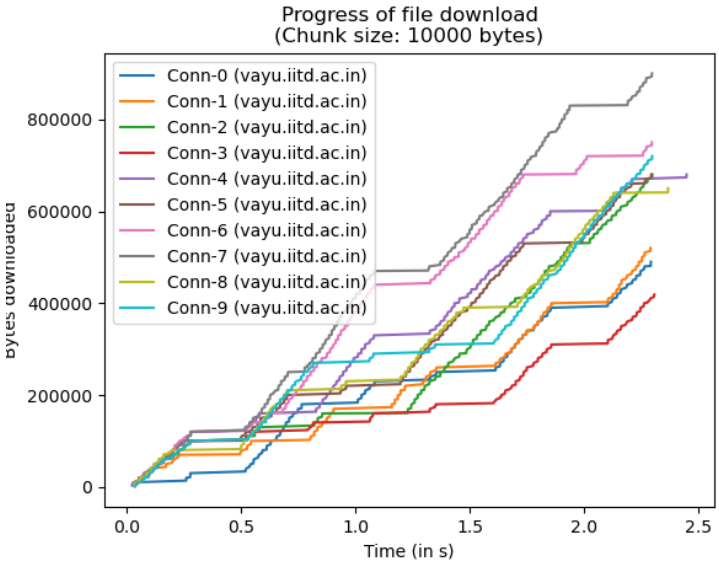

Figure 3: 10 connections

It is observed that the disparity in data downloaded among different connections increases as the number of connections increase, showing that some connections see more data download than others

Effect of varying chunk size

Table 2: Download time from vayu for different chunk sizes

No of connections	Time (s) for different chunk sizes				
	10k bytes	50k bytes	100k bytes	200k bytes	500k bytes
2	11.15	4.17	2.67	2.80	2.39
5	5.38	2.77	2.15	2.02	2.57
10	2.98	2.87	2.84	2.48	2.82

Thus it is observed that increasing the chunk size to a certain level leads to a decrease in download time. This is because, in the current implementation, there is an additional time of 1 RTT for every chunk downloaded, and increasing the chunk size would reduce this time, acting similar to pipelining.

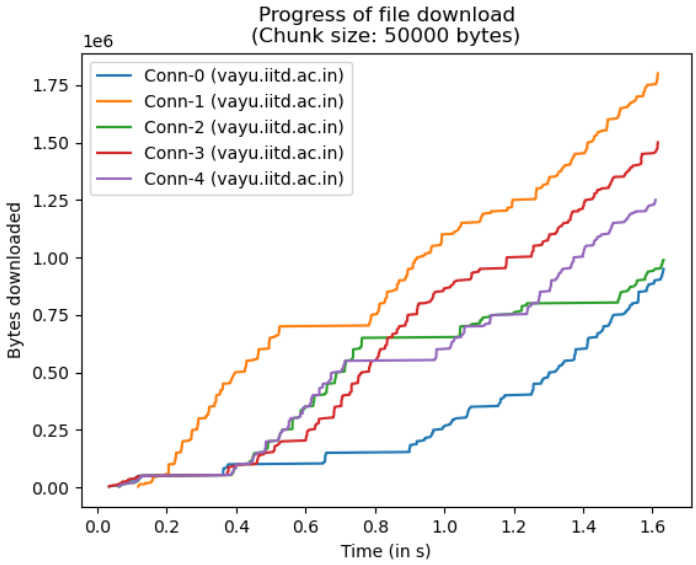

Figure 4: Chunk size 50k bytes, 5 connections

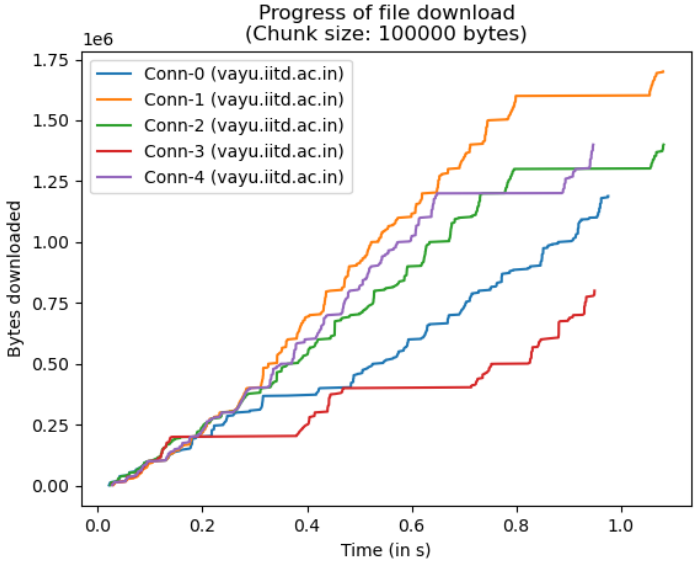

Figure 5: Chunk size 100k bytes, 5 connections

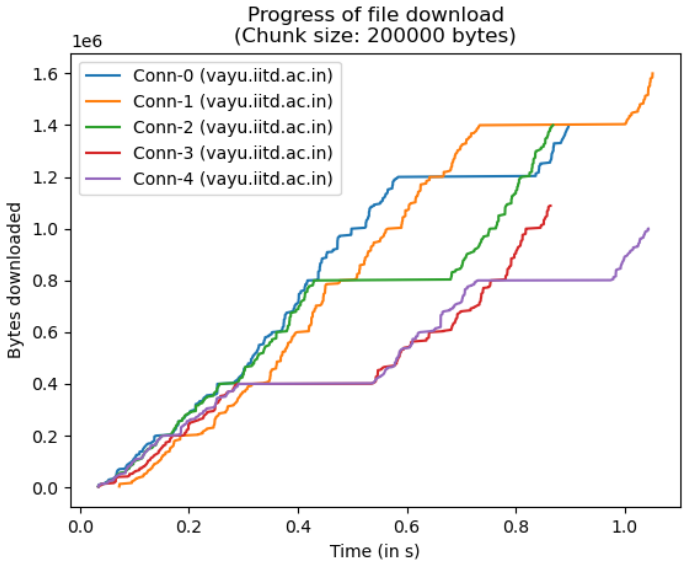

Figure 6: Chunk size 200k bytes, 5 connections

Effect of distributing load between different servers

Table 3: Download time for different no of connections to vayu and norvig for chunk size 100k bytes, keeping total no of connections as 5

Vayu connections	Norvig connections	Download time (s)
5	0	2.15
4	1	2.73
3	2	2.84
2	3	2.87
1	4	3.30
0	5	6.86

In general, downloading from multiple servers decreases overall time since we are freeing up load from one server (bottlenecks are located near the server's network interface). However, in this case it is observed that introducing connections to norvig increases the overall time, since from my location vayu has a much lower RTT (14ms) compared to norvig (300ms). Thus, in this case the bottleneck lies near the norvig server.

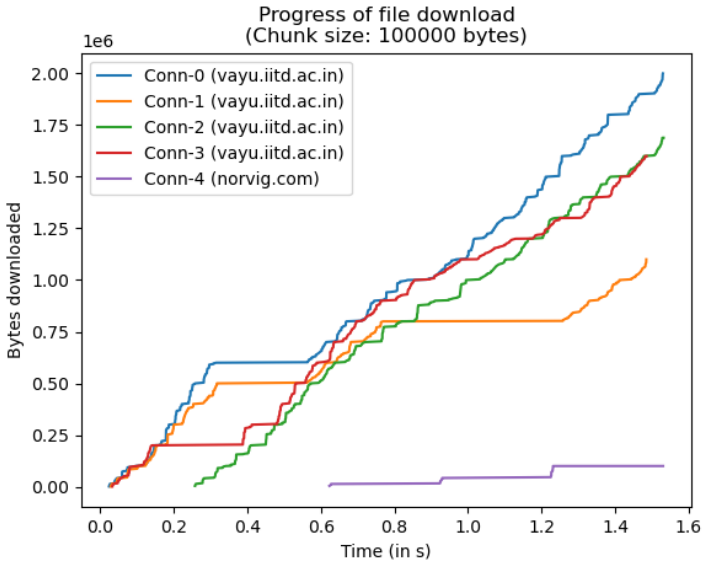

Figure 7: 4 vayu v/s 1 norvig connection

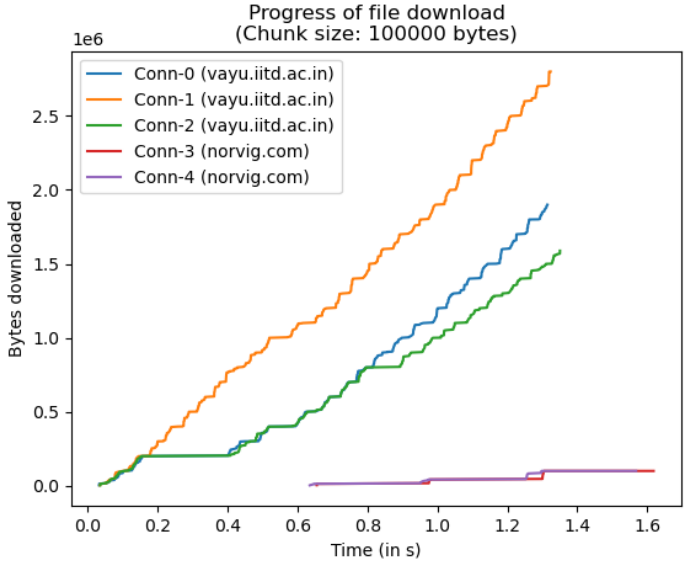

Figure 8: 3 vayu v/s 2 norvig connections

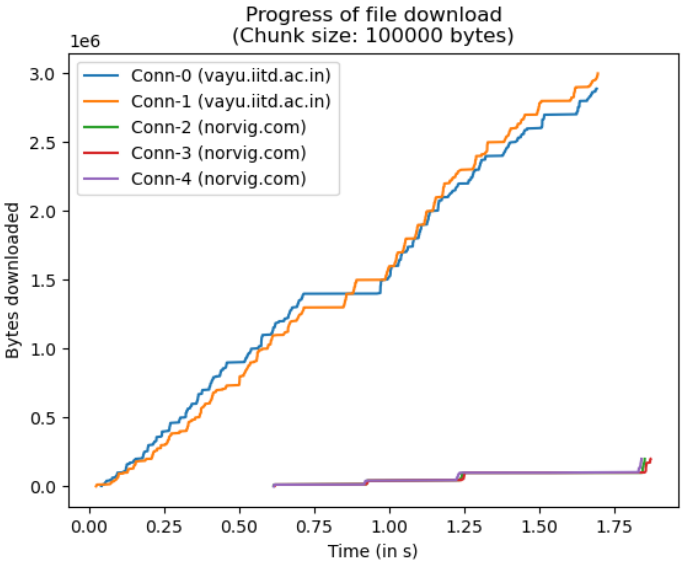

Figure 9: 2 vayu v/s 3 norvig connections

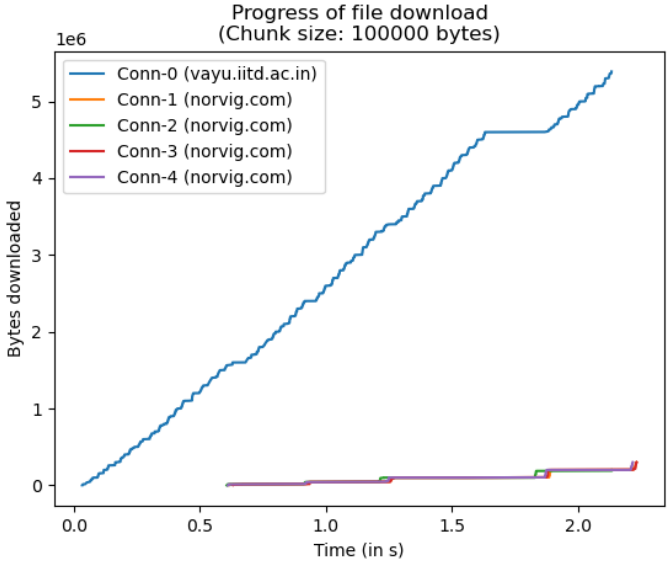

Figure 10: 1 vayu v/s 4 norvig connections

Thus it is observed that the downloaded bytes on connections to vayu are much more compared to connections to norvig, thus the script is able to utilise the difference in download speeds from the servers to download more from the faster server. This is because the threads downloading quickly also pick up new chunks quickly.

Question 4

To make the script resilient to network disconnections, the following is done

1. Try/except blocks are added to the function the threads are executing: in case an exception (like timeout) occurs while sending or receiving data, the exception is caught.

2. When an exception is caught, the existing socket is closed, and the try block is again entered. Here, a new socket is instantiated, which tries to establish a new connection.

3. On every exception, the script sleeps for 2 sec to prevent too many attempts at reconnection

4. Once a connection gets established, a new GET request is sent for the remaining part of the last chunk which was getting downloaded.

5. A timeout of 5sec is set on the sockets to detect network disconnections early

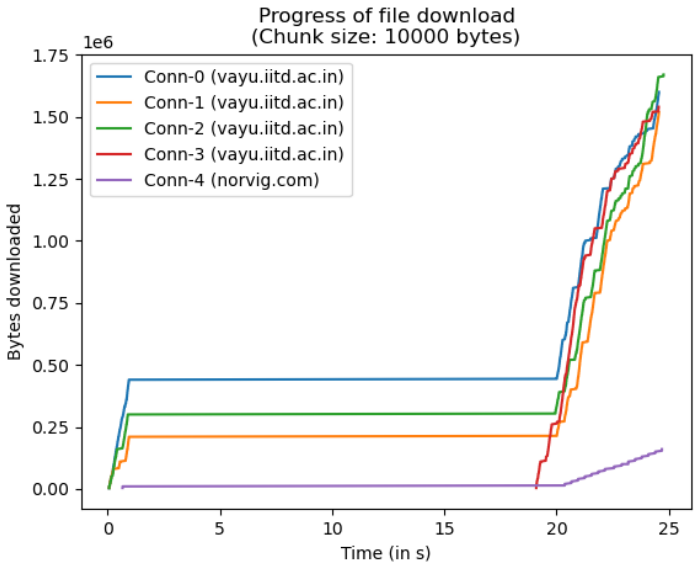

Figure 11: Introducing a disconnection in between

Thus the script stalls download on disconnection, and picks up download successfully once the connection is reestablished.