1

Implementation:

A.py contains the code for **part(a)**. In the first part we only needed to write a simple program to download the entire file using a single TCP connection and by raising a single GET request. I did all this in a file **A.py** in the code submitted. For this part I initialize a socket from client side using

```
sock=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

, then connected the host and port using

```
sock.connect((hostone,port))
```

Then, using socket initiated early, I just sent the following GET request using socket.send function:

```
b'GET /big.txt HTTP/1.1\r\nHOST: vayu.iitd.ac.in\r\n\r\n'
```

After which I used select.select as polling medium for 3 seconds, and then received data using

```
data = sock.recv(1)
```

Finally, splitted the data at using split function and then wrote the entire received bytes and checked for md5sum with original md5 provided.

Observation:

Total Time taken to download varied each time I ran the code:

Time taken in each run	
	Time Taken
1	17
2	34
3	18
4	26

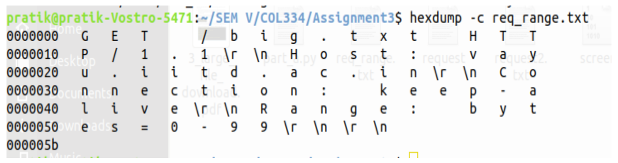
As this time whole file downloaded in one go, hence, the only bottleneck currently there is the download speed of the client. Other things like time taken to make TCP connection etc. are negligible as compared to that. Also, This reasoning also verifies the difference in the download time as well, because the download speed for the client varied.

2

B.py contains the code for **part(b)**. In this part we needed to append the Range parameter in sending GET command as well. New header file is mentioned below:

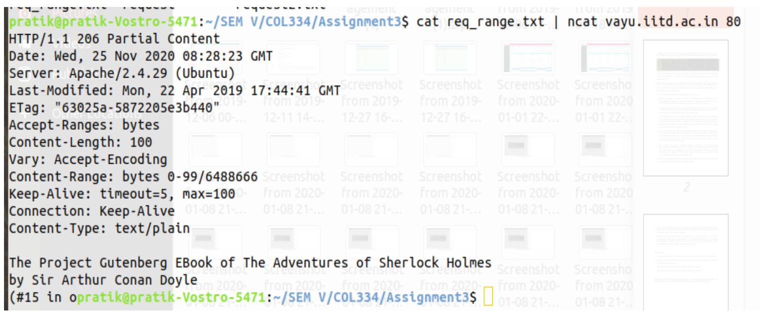
```
b'GET /big.txt HTTP/1.1\r\nHost: vayu.iitd.ac.in\r\nRange: bytes 0-99\r\n\r\n'
```

I tried the *cat* and *ncat* command to check whether the GET request sent is working perfectly used or not. Then I appended the same GET request to the file used in first question as well and the reply received indeed were the bytes asked in the GET command. Also, I checked for *hexdump* as well for the request which matched as well for the request.



```
pratik@pratik-Vostro-5471:~/SEM V/COL334/Assignment3$ hexdump -c req_range.txt
0000000   G   E   T       /   b   i   g   .   t   x   t       H   T   T
0000010   P   /   1   .   1  \r  \n   H   o   s   t   :       v   a   y
0000020   u   .   i   i   t   d   .   a   c   .   i   n  \r  \n   C   o
0000030   n   n   e   c   t   i   o   n   :       k   e   e   p   -   a
0000040   l   i   v   e  \r  \n   R   a   n   g   e   :       b   y   t
0000050   e   s   =   0   -   9   9  \r  \n  \r  \n
000005b
```

Also, following is the output of the *ncat* command:



```
pratik@pratik-Vostro-5471:~/SEM V/COL334/Assignment3$ cat req_range.txt | ncat vayu.iitd.ac.in 80
HTTP/1.1 206 Partial Content
Date: Wed, 25 Nov 2020 08:28:23 GMT
Server: Apache/2.4.29 (Ubuntu)
Last-Modified: Mon, 22 Apr 2019 17:44:41 GMT
ETag: "63025a-5872205e3b440"
Accept-Ranges: bytes
Content-Length: 100
Vary: Accept-Encoding
Content-Range: bytes 0-99/6488666
Keep-Alive: timeout=5, max=100
Connection: Keep-Alive
Content-Type: text/plain

The Project Gutenberg EBook of The Adventures of Sherlock Holmes
by Sir Arthur Conan Doyle
(#15 in opratik@pratik-Vostro-5471:~/SEM V/COL334/Assignment3$
```

3 & 4

C.py and **D.py** contains the code for **part(c)** and **part(d)** respectively. **In part(c)** we needed to maintain some TCP connections, download entire file in chunks and finally check whether md5 matches or not. Then, to spread the connection to download some chunks from one server and some chunks from other. In **part(d)** we needed to restructure program to handle for internet connectivity issue.

Implementation:

Idea:
What I did in this part is to read the number of TCP connections from the CSV file and the required object file. Then, initiate the same number of threads in my program as no. of TCP connection, then each thread starts a new connection with a socket.
I have done pipelining as well which means I initially send all the requests at once, then receive one by one. If it happens either connection dies out either due to exceeding limits of request, or there is a timeout of 5 sec, or even there is network issue, I then check the receive buffer, to just get all the request got till now, and then, similarly, recurse similarly on the remaining chunks didn't received till now.
I have used a queue for storing all the requests at a time, and then checked from the queue whether all the requests sent till now by thread have received or not, else resend all the requests again.

Optimization:
I have tried to assign each thread their own chunks so as to get rid of the time delay caused due to assignment of locks for writing on the final storing array. This assumption of assigning independent chunks to each of the thread also helps us get rid of the most important problem which occurs during pipeline i.e., multiple download of the same chunks.
Also, as each of the thread have their own independent chunks they could write in parallel as well on data array(used for bookkeeping).

Handling Errors:
Each of functions except *main*() all the functioning code is in try block. Hence, each time the functioning code raises an exception, Except block of code is called and each except block have their own respective error handling mechanism as described below.
Error handling for `sock.connect((hostone,port))`:

```
print('Socket connection error: ' + str(msg))
time.sleep(t)
print('Trying Again in ' + str(t) + ' sec')
connect_socket(sock,hostone,port,2*t)
```

The above code block is for error handling for *socket.connect*() function. Here, I have tried, ever there is an error, mostly, occurs due to internet connection issues, I went for multiplicative time increase for retrying,

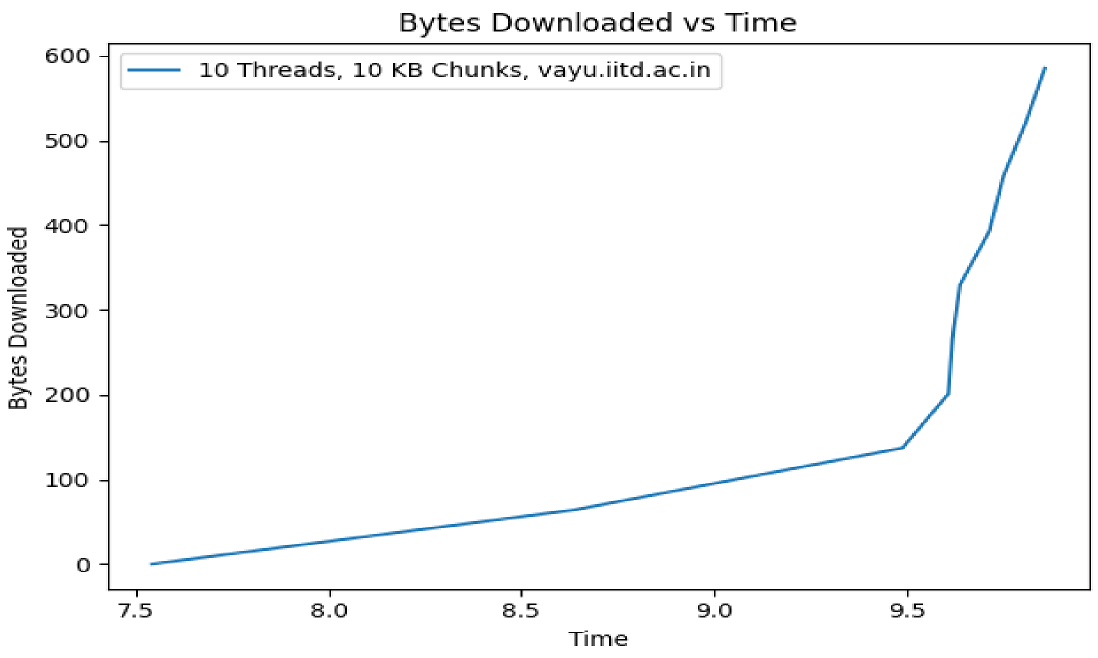
as connection issues in real life could be highly ambiguous.
Error handing for `sock.sendall(send_msg(st,en))` and `data=sock.recv(10240)`:

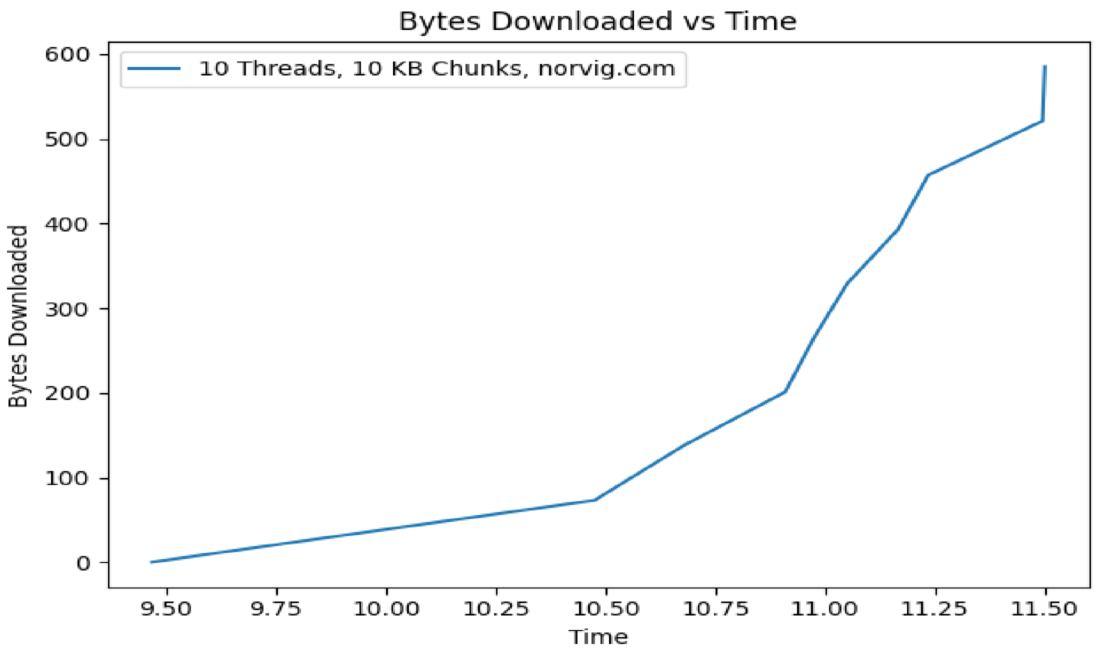
```
    close_socket(sock)
    print('Waiting for 1 sec then restarting')
    time.sleep(1)
    thread_op(host,port,mt,startf,endf)
```

`thread_op` is the target function for each of the thread in my program, whose parameters include, *host*,*portnnumber*, *mt*(queue for chunk to be requested), *startf* and *endf* re the chunks range assigned to each of the threads where each of them have to process. I first, closed the current socket, before starting the new socket which is created inside the `thread_op` function as given below:
Thread target function:

```
    def thread_op(host,port,queue,st,en):
        sock=create_socket()
        connect_socket(sock,host,port,1)
        download_chunk(sock,queue,1,st,en)
```

Total number of requests sent by a socket is at max 100 or the connection will itself dies out in 5 secs as well. I addressed this problem each time there is a time out or request limits exceed, I just resend all the remaining queries by making a new socket.
If some how when we send a large number of requests from client side, or if large number of socket is connected to server side, it randomly reset the connection by posing Error No. 104, by sending a RST packet. I solved the first problem by bottleneck the number of request sent at once, also, in the second case I did check for the exception and just checked for all the queries answered and created a new socket, after closing current ones and then resending all the remaining queries.
Internet related queries has already been discussed earlier.
Observations:
Graphs on next page shows the variation of chunks downloaded vs time. Number of TCP connections used are 10 and chunk size is 10KB. The first graph is for $Vayu$ while the second graph is for $Norwig$. The graph shows the data for download of entire text. As you could see both the graphs starts at after a time and then the download is instaneous which is due to pipelined approach as I am querying all the request at once and then waiting for all of them to receive. Higher speed here is observed for $Vayu$ as compared to $Norwig$ reasoning of which is explained afterwards.


Bytes Downloaded vs Time
10 Threads, 10 KB Chunks, vayu.iitd.ac.in
Bytes Downloaded
Time
0
100
200
300
400
500
600
7.5
8.0
8.5
9.0
9.5


Bytes Downloaded vs Time
10 Threads, 10 KB Chunks, norvig.com
Bytes Downloaded
Time
0
100
200
300
400
500
600
9.50
9.75
10.00
10.25
10.50
10.75
11.00
11.25
11.50

Number of TCP Connections:

Following table shows TCP-connections vs Time data for both *norwig* and *vayu*.

Time in seconds		
TCP Connections	**norwig.com**	**vayu.iitd.ac.in**
1	63.99	32.83
2	36.41	17.59
4	19.92	10.63
8	10.97	7.35
16	9.78	7.88
32	12.93	7.56
64	10.09	8.91

We could easily observe that the time taken by $Vayu$ is very less than $Norwig$. In my opinion the relative time lag between $Vayu$ and $Norwig$ could be due to the increased RTT for $Norvig$ as compared to $Vayu$. We could easily confirm it from the traceroute output for both of them.
For $Vayu$ it is at max around 50 ms while for $Norvig$ it is around 400 ms at times.
Now, the second observation we could see is increased speed for getting required data with increased TCP connections which could be as with more TCP connection running on parallel threads data could be downloaded parallelly. Hence, there is a distribution of job between the threads and the latency almost decreases by half, for like going from 1 threads to 2 threads or from 2 threads to 4 threads. But after that increasing the connection seemed more redundant as now the bottleneck seemed to be making new connection as same amount of data could be downloaded from almost half of the threads in one go as well. So, we could easily observe there is an increase in latency in both $Vayu$ and $Norvig$ towards the end increase of TCP connections.

Following is the TCP-Connections vs time table for combined case.

Time in seconds	
TCP Connections	**combined(Norvig and Vayu)**
2	63.99
4	16.02
8	9.26
16	10.41
32	8.69
64	7.48

Here I have tried to combine the data download from both the $Norvig$ as well as $Vayu$. Here, I tried dividing the data download from both $Vayu$ as well as $Norvig$ by assigning the half of the threads for downloading from $Vayu$, and assigning half of the thread to download from $Norvig$. Here, we could observe the time is between $Norvig$ and $Vayu$ which is in some sense obvious as half of the chunks are getting downloaded from $Norvig$ while half of the chunk is downloaded from $Vayu$.