COL334 HW3

Rishabh Ranjan – 2018CS10416

November 25, 2020

1 Question 1

Following the instructions in the question carefully, I wrote the following Python-3.8 program:

```
import socket
import re
import tqdm

sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect(('norvig.com', 80))
sock.sendall(b'GET /big.txt HTTP/1.1\r\nHost: norvig.com\r\n\r\n')
clen_re = re.compile(br'\r\nContent-Length: (\d*)\r\n')

buf = b''
while True:
    buf += sock.recv(1)
    if buf[-4:] == b'\r\n\r\n':
        clen = int(clen_re.search(buf).group(1))
        break

with open('q1.txt', 'wb') as f:
    for _ in tqdm.trange(clen, total=clen, unit='b', unit_scale=True):
        f.write(sock.recv(1))
```

With this I was able to download the file correctly. I verified the MD5 sum with:

```
md5sum q1.txt
```

2 Question 2

Again, I was able to download parts of the file by following the given instructions. To insert `\r` characters I used `Ctrl+V Ctrl+M` in the `INSERT` mode. With `vim -b`, `\r` can be seen as `^M`. The hexdump matched the one given in the assignment description exactly. The received response is shown below.

```
HTTP/1.1 206 Partial Content
Date: Wed, 25 Nov 2020 13:10:57 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 o
```

3 Question 3

3.1 A note on implementation: asyncio module

I have used Python's `asyncio` module for this assignment. `asyncio` offers a model of concurrency known as *cooperative multitasking*, which is a recommended alternative to threads for *IO-bound concurrency* such as networking. The module is fairly new and gaining popularity. I have used Python-3.8, using the latest update to the `asyncio` module in Python-3.7.

Threads in python do not really offer any parallelism due to limitations imposed by the Python GIL (Global Interpreter Lock). `asyncio` is a single-process, single-thread paradigm which uses an event-loop and callback mechanism to orchestrate many co-routines simultaneously and efficiently. Particularly it has a **very low overhead** compared to both multi-threading and multi-processing in Python. Also, it is simpler to use and designed to handle network related tasks well where the bottleneck is the network and not the CPU. With lower overheads, the trends with multiple TCP connections are clearer. For these and other reasons in favour of `asyncio`, I felt it would be a good design decision to use `asyncio` instead of threads to manage parallel TCP connections.

I have confirmed that this would be allowed in this assignment without any penalty in marks through a private Piazza post (answered by Aaditeshwar Sir himself).

3.2 Motivation

The initial strategy I experimented with was to divide the full content length into as many chunks as the number of parallel TCP connections. Then on each TCP connection an entire chunk was requested at once. This approach shines when all the TCP connections are to the same server which is capable of supporting all the connections equally well. In this case, all connections transfer data at almost equal rates so all responses complete in similar times without leaving any connection idle for long. Moreover, since all data is requested by a single GET request on a connection, we avoid issues like pipelining and can expect faster transfer than with multiple GET requests where the server must process each GET request separately. This approach does not compromise on robustness because data from a response is received in parts (dictated by the network buffer) and is guaranteed to be in order. So, if a connection is lost even before the response was complete, the data received so far can be saved and a new connection can easily resume downloading the remaining data by computing appropriate byte offsets.

However, the above strategy has a flaw which becomes most apparent when parallel TCP connections to multiple servers are available and some servers are slower than the others. Because the above strategy statically assigns the byte ranges to download to the different TCP connections (assuming equal download speeds), it may so happen that the faster server has completed it's requests and is idle (or has closed it's connections) while the slower server is still going on very slowly.

It is immediately evident that this issue can be resolved by dynamic load distribution on the parallel TCP connections available. To do this, we send new requests over TCP connections which complete previous requests earlier. We cannot pre-assign portions of the file to TCP connections here. However, we still want to request for sufficient data in each request so that the overall system is efficient. For this we split the file into chunks and request chunks as a whole over TCP connections which have completed earlier requests.

This system can adapt to the download speeds of the different servers/connections.

Since we are guaranteed that the responses to multiple GET requests will arrive in order, we do not need to wait to receive the full response to the previous request before sending new ones as that would be inefficient. So, we maintain a pipeline where multiple GET requests have already been sent on the connection after the GET request which is currently being responded to.

3.3 Implementation details

3.3.1 Dependencies

Python-3.8 and `tqdm` module.

3.3.2 Usage

```
usage: dl.py [-h] [-c CHUNK_SIZE] [-p PIPELINE] [-t TIMEOUT] [-r RETRY_GAP] [-s PLOT_PATH]
csv_path out_path

positional arguments:
  csv_path              input CSV file with URLs and number of parallel TCP connections to use
  out_path              output file for download

optional arguments:
  -h, --help            show this help message and exit
  -c CHUNK_SIZE, --chunk_size CHUNK_SIZE
                        chunk size in bytes
  -p PIPELINE, --pipeline PIPELINE
                        number of GET requests to pipeline
  -t TIMEOUT, --timeout TIMEOUT
                        TCP timeout
  -r RETRY_GAP, --retry_gap RETRY_GAP
                        time to retry connection after
  -s PLOT_PATH, --plot_path PLOT_PATH
                        draw and store plot in path
```

3.3.3 Brief overview

Details can be found by examining the code and accompanying comments. Here I provide a brief overview:

- A HEAD request is sent first to get the content length.
- An object `range_gen` is constructed to manage download tasks. Specifically, when queried it offers a bytes range (chunk) which has not yet been downloaded.
- A file is allocated with size same as content length (efficient disk IO on large files).
- Parallel TCP connections are created following the CSV file.
- At the start of each connection a number of GET requests are sent and then whenever a GET request is completed a new one is added if possible. This simulates a **pipeline** with desired capacity.
- When a chunk is received completely, it is stored into the output file at the correct byte offset.
- The program finishes when all the bytes of the file have been received.
- Broken connections are also handled. See section Question 4 for details.

3.4 Observations

I observed three configurations for comparison:

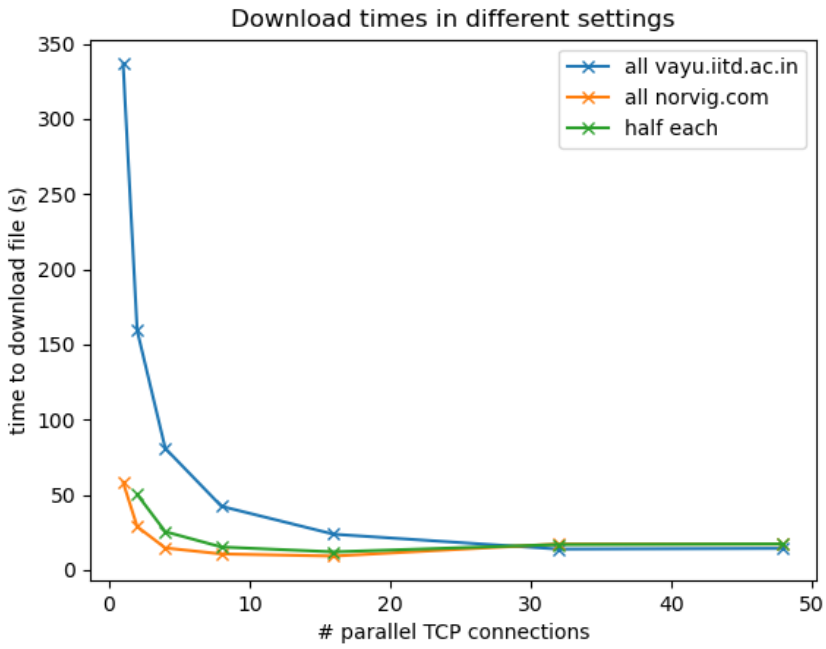
1. all `vayu.iitd.ac.in`: all TCP connections are to `vayu.iitd.ac.in`
2. all `norvig.com`: all TCP connections are to `norvig.com`
3. half each: half of the TCP connections are to `vayu.iitd.ac.in` and half are to `norvig.com`

The results are discussed below.

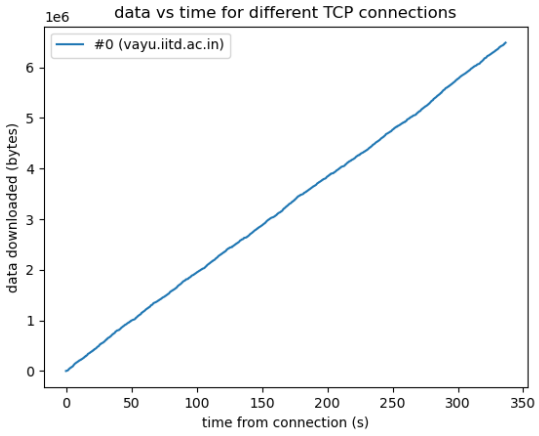
3.4.1 Table of download times

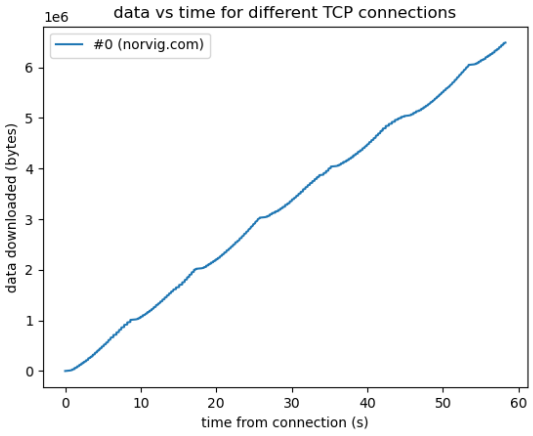
# parallel TCP connections	all `vayu.iitd.ac.in` (s)	all `norvig.com` (s)	half each (s)
1	336.53	58.28	-
2	159.28	28.86	50.28
4	80.91	14.66	25.49
8	42.36	10.68	15.29
16	23.79	9.27	12.04
32	13.91	17.32	16.94
48	14.36	17.31	17.34

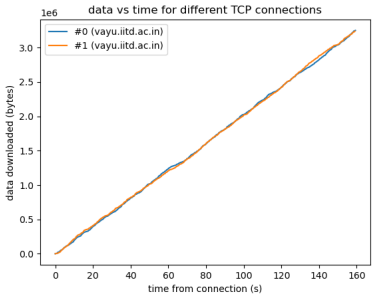
3.4.2 Plot of download times

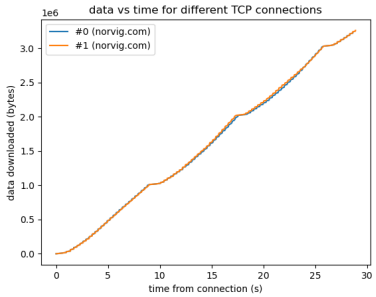


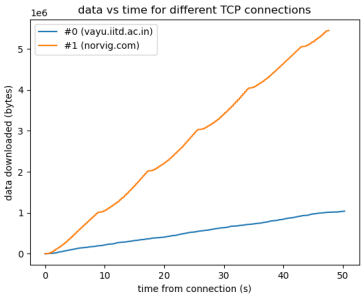
3.5 Plots of download progress

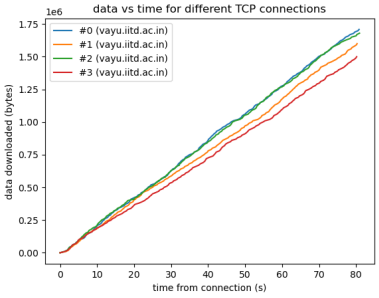


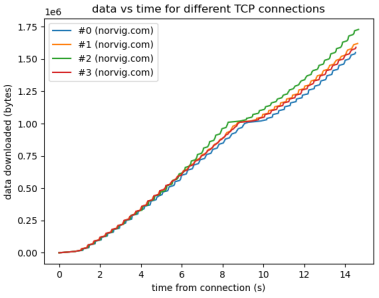


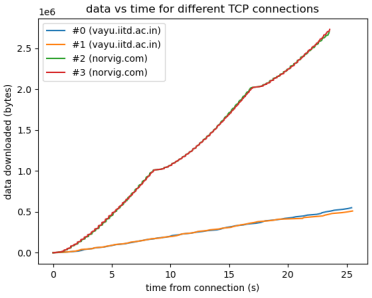


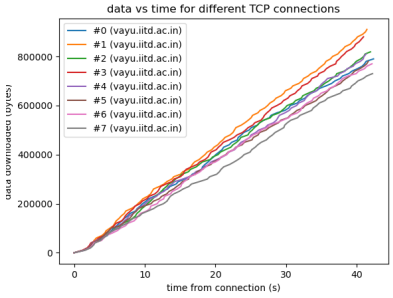


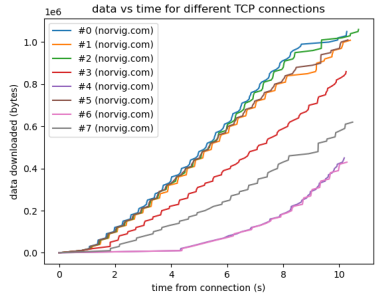


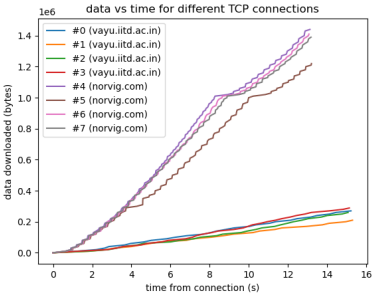


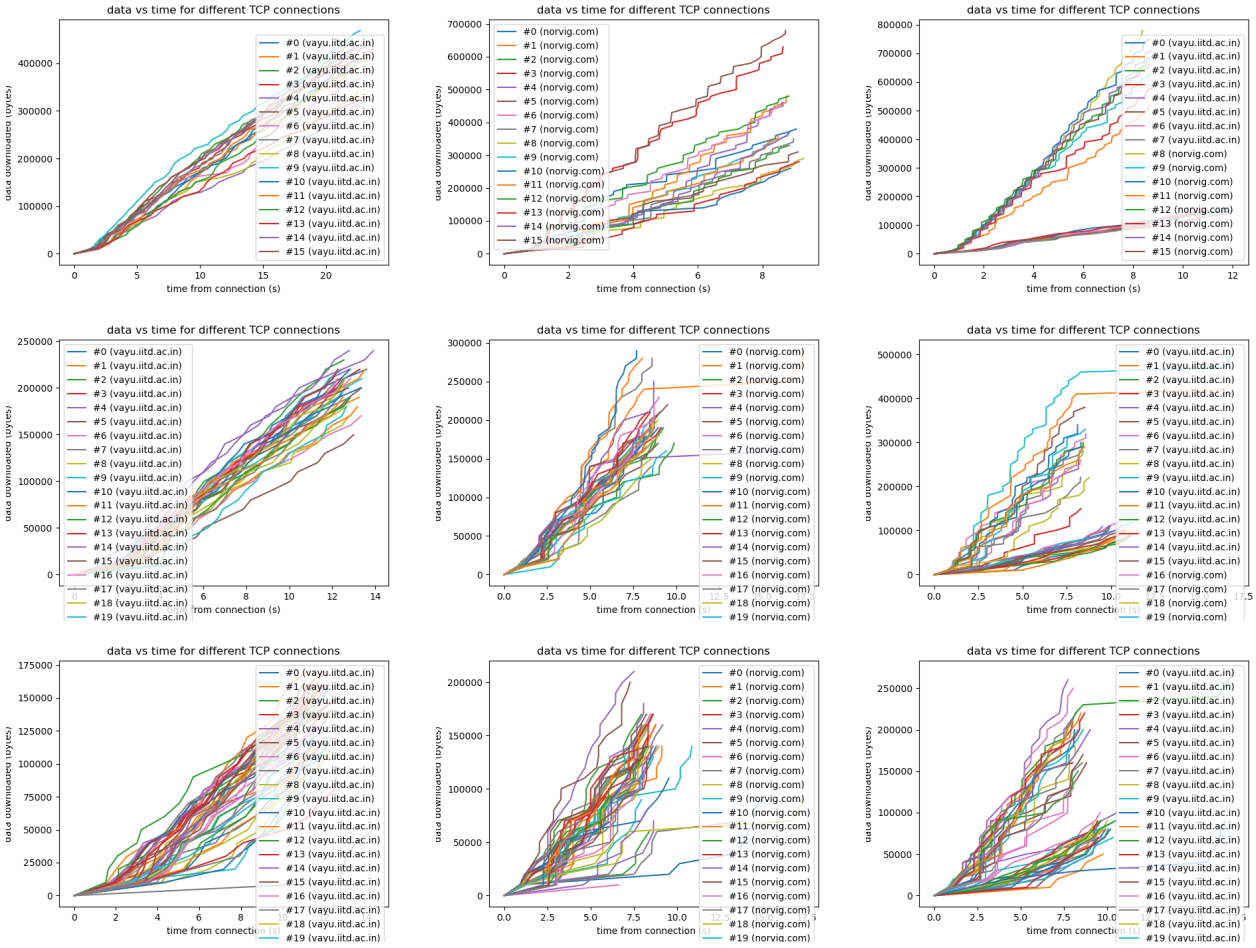












3.6 Discussion

We observe that when the number of parallel TCP connections increases, the download time decreases initially. This is most pronounced for smaller number of TCP connections. As the number of connections increases to large values, the download time saturates and even tends to increase slightly.

The explanation for this behavior lies in the **bandwidth delay product** of the connection which is the product of the *bandwidth* and the *round trip time*. The bandwidth delay product essentially measure the amount of data in-flight in the network, i.e the amount of data that will be sent before an acknowledgement is received. In high-latency, high-bandwidth environments like the internet, the data transfer rates of a reliable protocol such as TCP is limited by the bandwidth delay product. The bandwidth delay product is defined per connection so having multiple connections allows to circumvent it and data transfer may benefit from it.

The bandwidth delay product is governed by the **TCP receive window** which is often too small to make use of the full end-to-end network bandwidth. However, with multiple connections this can be overcome because different TCP connections have their own *send/receive buffers* and hence their own TCP receive windows.

In this way by increasing the number of connections we may be able to utilise the full network bandwidth available to us. This leads to the **saturation** observed at higher number of connections. However, there are caveats to this. The speed, buffer sizes, congestion, serving policy etc. of the server or the network may prevent the data transfer rate from utilising the full network bandwidth so the saturation can happen

pre-maturely. Further, more connections impose higher overheads and may slightly increase the times after saturation.

Another factor is that the server may try to fairly use its resources to serve the connections it has accepted. However, by establishing multiple connections to a busy server we are able to compete for a larger portion of the server's "attention".

It should also be noted that the data trends reported here are conditional on the particular situation in which they were collected. Particularly, both these servers seemed to be busy at the time of data collection, probably due to traffic from COL334 students trying to complete this assignment! On previous runs I observed that `vayu.iitd.ac.in` is faster than `norvig.com` which can be explained by the geographical distances to these servers. However, under loaded conditions, `norvig.com` responded much better. Also, the effect of multiple connections was more pronounced because the servers were busy as new connections made significant impact in terms of resource sharing.

From the download progress plots we see that for the same server, all connections show similar progress. Any significant *stalling* was not observed. The download progress has a zig-zag step-like shape which is simply because of **network buffering** and **multitasking** between various connections due to which data becomes available in bursts and remains somewhat idle in between.

When the download is spread over additional connections to another server, the download time is generally seen to decrease. Again this is expected for the same reasons as discussed above. This shows that the *bottlenecks* lie in the individual TCP connections and the servers serving on them, at least for sufficiently few parallel connections. Further, we observe that the times for using both servers lie between the time for using each server alone. However, due to dynamic load balancing the time is much closer to the one for the faster server.

In my runs, `norvig.com` server was a lot faster than the `vayu.iitd.ac.in` server. The program was easily able to adapt to this as can also be seen from the plots – the total data downloaded via the `norvig.com` connections is a lot more than the total data downloaded via the `vayu.iitd.ac.in` connections.

4 Question 4

To detect broken connections, I use a scheduled callback `check_connection` after every `timeout` seconds to check whether any data was received on that connection in the time interval from the last `check_connection` (which is roughly equal to `timeout`).

When it is detected that a connection has been lost, I close the connection from my end. The chunks in the pipeline of this connection are flushed back into the pool of remaining chunks. If there are remaining chunks, we keep trying to reconnect. When a connection succeeds it behaves as a fresh connection and resumes the download.

I observed that the default TCP timeout is large (approximately **15 minutes**). However, in practical cases the server closes the connection earlier than this. It may be possible that the closing messages of the server was lost in the time when the network was disconnected. So, if we rely on a closing message we may never receive one and the TCP connection from our side will remain stuck in a limbo. This makes it important to have active checks for a lost connection.

Testing with network disconnections shows that the program is able to resume download when the connection is available again. Further, this does not affect the correctness of the downloaded file, which I verified using the MD5 sum.