Name: Meetakshi Setiya
Roll no: 2019253

CSE-232 Computer Networks

Assignment 2

Q1.

System: Ubuntu 18.04 virtual machine
Network: WiFi
Protocol: IPv4

The server-client program works this way- each client client first sends a number N (5, here) to the server- marking that it wants the server to send its top N CPU consuming processes to it- which is 5 in the code I am running. After this, the server sends its top 5 processes to the client and the client in turn sends its highest CPU consuming process to the client. This happens for each client that connects with the server.

I used a bash script script.sh to run two new concurrent client process every second to send continuous concurrent connection requests to to the server. I then used the timeout command to automatically terminate the script after 2 minutes.

The shell script is:

```
#!/bin/sh
while true
do
   ./client &
   ./client &
   sleep 1
done
```

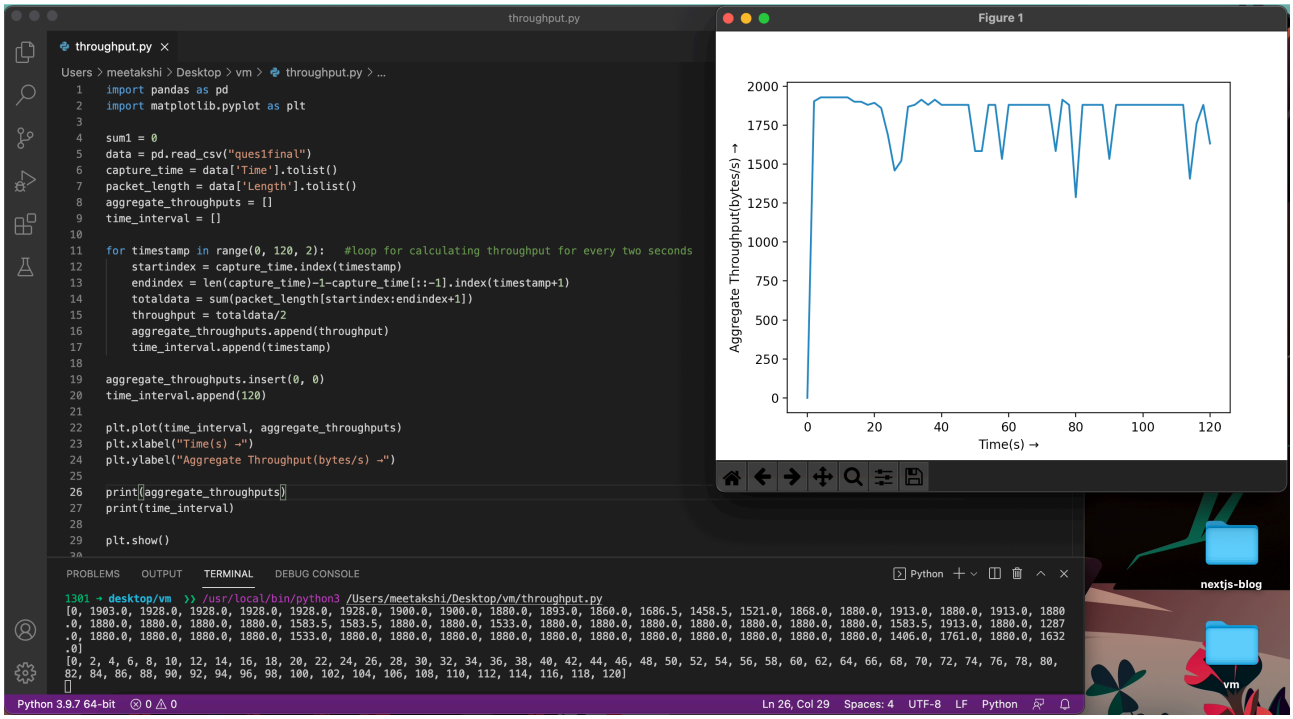
The timeout command is: `timeout 2m bash script.sh`

I collected the packet trace using Wireshark for loopback interface, at tcp port 8888 (capture filter- `tcp port 8888`). This just captures the traffic to and from port 8888- where my server is running. To remove the DNS cache, I again used a display filter `ip.addr == 127.0.0.1 and tcp.port == 8888`, just to make sure that I have the packets for the client server socket program only. There were a total of 2784 packets after filtering for the client-server tcp socket program packets. The uplink and downlink traffic was captured.
The time was collected in seconds to enable easy calculation of per 2 second throughput.

The first few packets out of the entire packet trace looked like this (the time is in seconds):

No.	Time	Source	Destination	Protocol	Length	Info
1	0	127.0.0.1	127.0.0.1	TCP	74	58086 → 8888 [SYN] Seq=0 Win=65495 Len=0 MSS=65495 SACK_PERM=1 TSval=315…
2	0	127.0.0.1	127.0.0.1	TCP	74	8888 → 58086 [SYN, ACK] Seq=0 Ack=1 Win=65483 Len=0 MSS=65495 SACK_PERM=…
3	0	127.0.0.1	127.0.0.1	TCP	66	58086 → 8888 [ACK] Seq=1 Ack=1 Win=65536 Len=0 TSval=3153893767 TSecr=31…
4	0	127.0.0.1	127.0.0.1	TCP	74	58088 → 8888 [SYN] Seq=0 Win=65495 Len=0 MSS=65495 SACK_PERM=1 TSval=315…
5	0	127.0.0.1	127.0.0.1	TCP	74	8888 → 58088 [SYN, ACK] Seq=0 Ack=1 Win=65483 Len=0 MSS=65495 SACK_PERM=…
6	0	127.0.0.1	127.0.0.1	TCP	66	58088 → 8888 [ACK] Seq=1 Ack=1 Win=65536 Len=0 TSval=3153893771 TSecr=31…
7	0	127.0.0.1	127.0.0.1	TCP	67	58086 → 8888 [PSH, ACK] Seq=1 Ack=1 Win=65536 Len=1 TSval=3153893782 TSe…
8	0	127.0.0.1	127.0.0.1	TCP	66	8888 → 58086 [ACK] Seq=1 Ack=2 Win=65536 Len=0 TSval=3153893782 TSecr=31…
9	0	127.0.0.1	127.0.0.1	TCP	67	58088 → 8888 [PSH, ACK] Seq=1 Ack=1 Win=65536 Len=1 TSval=3153893797 TSe…
10	0	127.0.0.1	127.0.0.1	TCP	66	8888 → 58088 [ACK] Seq=1 Ack=2 Win=65536 Len=0 TSval=3153893797 TSecr=31…
11	0	127.0.0.1	127.0.0.1	TCP	176	8888 → 58086 [PSH, ACK] Seq=1 Ack=2 Win=65536 Len=110 TSval=3153893998 T…
12	0	127.0.0.1	127.0.0.1	TCP	66	58086 → 8888 [ACK] Seq=2 Ack=111 Win=65536 Len=0 TSval=3153893998 TSecr=…
13	0	127.0.0.1	127.0.0.1	TCP	181	8888 → 58088 [PSH, ACK] Seq=1 Ack=2 Win=65536 Len=115 TSval=3153894006 T…
14	0	127.0.0.1	127.0.0.1	TCP	66	58088 → 8888 [ACK] Seq=2 Ack=116 Win=65536 Len=0 TSval=3153894006 TSecr=…
15	0	127.0.0.1	127.0.0.1	TCP	91	58086 → 8888 [PSH, ACK] Seq=2 Ack=111 Win=65536 Len=25 TSval=3153894006 …
16	0	127.0.0.1	127.0.0.1	TCP	66	8888 → 58086 [ACK] Seq=111 Ack=27 Win=65536 Len=0 TSval=3153894006 TSecr…
17	0	127.0.0.1	127.0.0.1	TCP	66	58086 → 8888 [FIN, ACK] Seq=27 Ack=111 Win=65536 Len=0 TSval=3153894006 …
18	0	127.0.0.1	127.0.0.1	TCP	66	8888 → 58086 [FIN, ACK] Seq=111 Ack=28 Win=65536 Len=0 TSval=3153894006 …
19	0	127.0.0.1	127.0.0.1	TCP	66	58086 → 8888 [ACK] Seq=28 Ack=112 Win=65536 Len=0 TSval=3153894006 TSecr…
20	0	127.0.0.1	127.0.0.1	TCP	91	58088 → 8888 [PSH, ACK] Seq=2 Ack=116 Win=65536 Len=25 TSval=3153894028 …
21	0	127.0.0.1	127.0.0.1	TCP	66	8888 → 58088 [ACK] Seq=116 Ack=27 Win=65536 Len=0 TSval=3153894028 TSecr…
22	0	127.0.0.1	127.0.0.1	TCP	66	8888 → 58088 [FIN, ACK] Seq=116 Ack=27 Win=65536 Len=0 TSval=3153894028 …
23	0	127.0.0.1	127.0.0.1	TCP	66	58088 → 8888 [FIN, ACK] Seq=27 Ack=117 Win=65536 Len=0 TSval=3153894031 …
24	0	127.0.0.1	127.0.0.1	TCP	66	8888 → 58088 [ACK] Seq=117 Ack=28 Win=65536 Len=0 TSval=3153894031 TSecr…
25	1	127.0.0.1	127.0.0.1	TCP	74	58090 → 8888 [SYN] Seq=0 Win=65495 Len=0 MSS=65495 SACK_PERM=1 TSval=315…
26	1	127.0.0.1	127.0.0.1	TCP	74	8888 → 58090 [SYN, ACK] Seq=0 Ack=1 Win=65483 Len=0 MSS=65495 SACK_PERM=…
27	1	127.0.0.1	127.0.0.1	TCP	66	58090 → 8888 [ACK] Seq=1 Ack=1 Win=65536 Len=0 TSval=3153894804 TSecr=31…
28	1	127.0.0.1	127.0.0.1	TCP	67	58090 → 8888 [PSH, ACK] Seq=1 Ack=1 Win=65536 Len=1 TSval=3153894810 TSe…
29	1	127.0.0.1	127.0.0.1	TCP	66	8888 → 58090 [ACK] Seq=1 Ack=2 Win=65536 Len=0 TSval=3153894810 TSecr=31…
30	1	127.0.0.1	127.0.0.1	TCP	74	58092 → 8888 [SYN] Seq=0 Win=65495 Len=0 MSS=65495 SACK_PERM=1 TSval=315…
31	1	127.0.0.1	127.0.0.1	TCP	74	8888 → 58092 [SYN, ACK] Seq=0 Ack=1 Win=65483 Len=0 MSS=65495 SACK_PERM=…
32	1	127.0.0.1	127.0.0.1	TCP	66	58092 → 8888 [ACK] Seq=1 Ack=1 Win=65536 Len=0 TSval=3153894830 TSecr=31…
33	1	127.0.0.1	127.0.0.1	TCP	67	58092 → 8888 [PSH, ACK] Seq=1 Ack=1 Win=65536 Len=1 TSval=3153894839 TSe…
34	1	127.0.0.1	127.0.0.1	TCP	66	8888 → 58092 [ACK] Seq=1 Ack=2 Win=65536 Len=0 TSval=3153894839 TSecr=31…
35	1	127.0.0.1	127.0.0.1	TCP	186	8888 → 58090 [PSH, ACK] Seq=1 Ack=2 Win=65536 Len=120 TSval=3153895022 T…
36	1	127.0.0.1	127.0.0.1	TCP	66	58090 → 8888 [ACK] Seq=2 Ack=121 Win=65536 Len=0 TSval=3153895022 TSecr=…
37	1	127.0.0.1	127.0.0.1	TCP	91	58090 → 8888 [PSH, ACK] Seq=2 Ack=121 Win=65536 Len=25 TSval=3153895027 …

I then exported the data from Wireshark to a CSV file and used a python program to calculate the aggregate throughput for ever 2 seconds. The code used is given as a



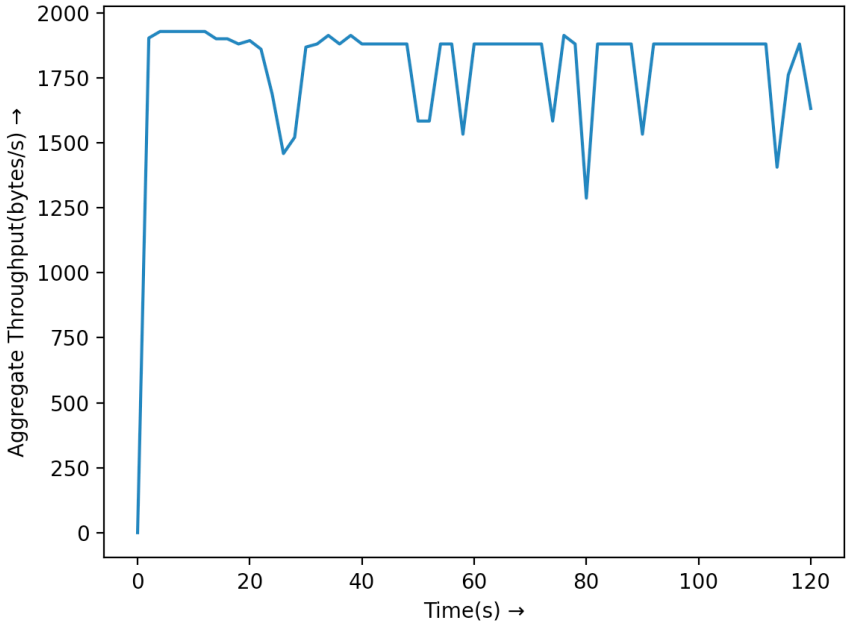

```
import pandas as pd
import matplotlib.pyplot as plt

sum1 = 0
data = pd.read_csv("ques1final")
capture_time = data['Time'].tolist()
packet_length = data['Length'].tolist()
aggregate_throughputs = []
time_interval = []

for timestamp in range(0, 120, 2):   #loop for calculating throughput for every two seconds
    startindex = capture_time.index(timestamp)
    endindex = len(capture_time)-1-capture_time[::-1].index(timestamp+1)
    totaldata = sum(packet_length[startindex:endindex+1])
    throughput = totaldata/2
    aggregate_throughputs.append(throughput)
    time_interval.append(timestamp)

aggregate_throughputs.insert(0, 0)
time_interval.append(120)

plt.plot(time_interval, aggregate_throughputs)
plt.xlabel("Time(s) →")
plt.ylabel("Aggregate Throughput(bytes/s) →")

print(aggregate_throughputs)
print(time_interval)

plt.show()
```

```
1301 → desktop/vm  ❯❯ /usr/local/bin/python3 /Users/meetakshi/Desktop/vm/throughput.py
[0, 1903.0, 1928.0, 1928.0, 1928.0, 1928.0, 1928.0, 1900.0, 1900.0, 1880.0, 1893.0, 1860.0, 1686.5, 1458.5, 1521.0, 1868.0, 1880.0, 1913.0, 1880.0, 1913.0, 1880
.0, 1880.0, 1880.0, 1880.0, 1880.0, 1583.5, 1583.5, 1880.0, 1880.0, 1533.0, 1880.0, 1880.0, 1880.0, 1880.0, 1880.0, 1880.0, 1880.0, 1583.5, 1913.0, 1880.0, 1287
.0, 1880.0, 1880.0, 1880.0, 1880.0, 1533.0, 1880.0, 1880.0, 1880.0, 1880.0, 1880.0, 1880.0, 1880.0, 1880.0, 1880.0, 1880.0, 1406.0, 1761.0, 1880.0, 1632
.0]
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62, 64, 66, 68, 70, 72, 74, 76, 78, 80,
82, 84, 86, 88, 90, 92, 94, 96, 98, 100, 102, 104, 106, 108, 110, 112, 114, 116, 118, 120]
```

screenshot below along with the accompanying outputs.

I also noticed that including the SYN and ACK packets, there were 12 packets exchanged between each client process with the server.

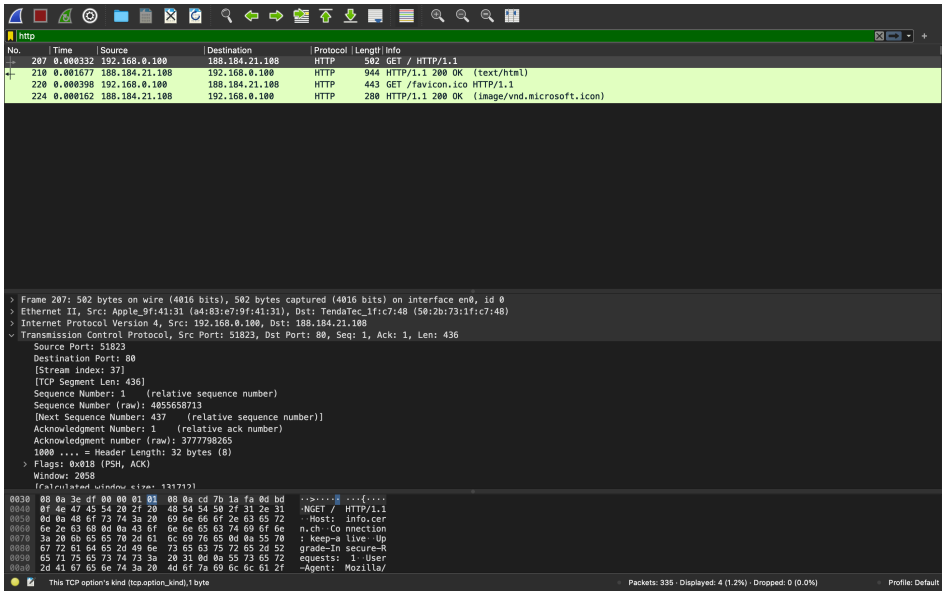
The aggregate throughput calculated for every 2 seconds against time:



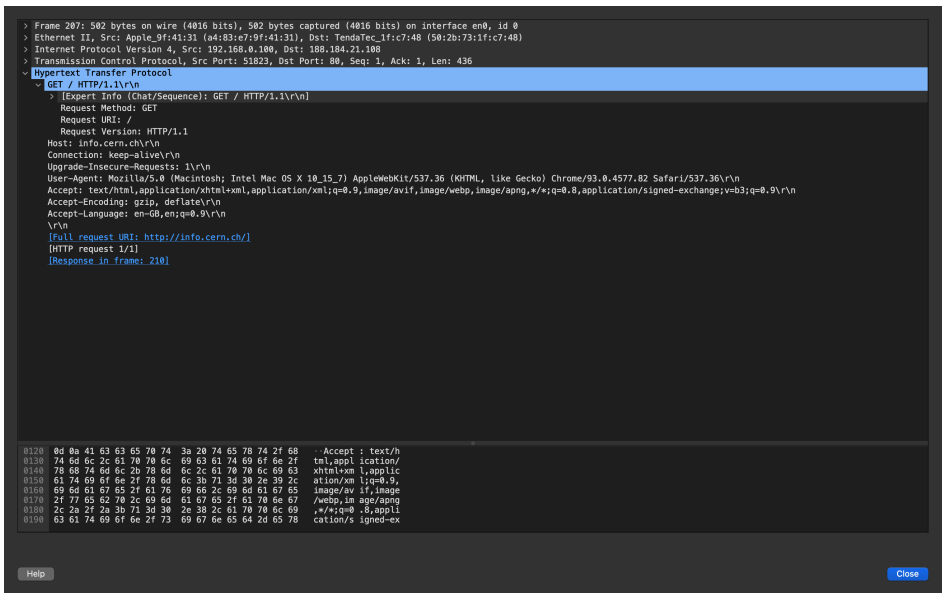
Q2.

System: MacOS Big Sur
Browser: Google Chrome
Packet analyser: Wireshark

Screenshot of all the packets captured while retrieving http://info.cern.ch is below:



1. **Packet no. 207 (as per the screenshot above):**



• HTTP packet type - `Request`
Here, my web browser (client) has sent an HTTP request to initiate an action on the server.

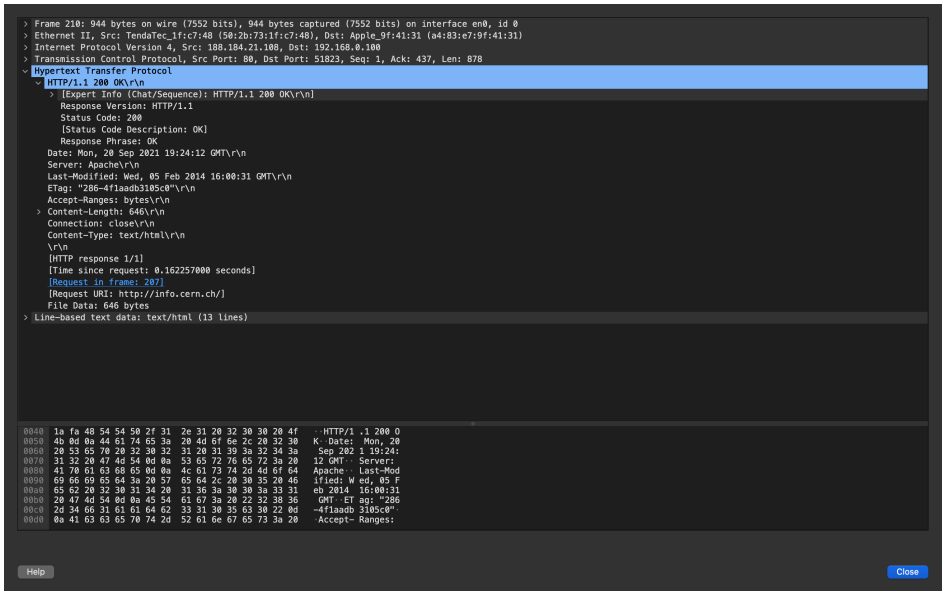
• HTTP request type - `GET`
The web browser (client) sent a HTTP GET request to the web server to request the HTML (text) on the webpage.

• User agent type - `Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/93.0.4577.82 Safari/537.36`
I used Google Chrome to retrieve the given web page and I am working on an Apple Laptop. The user agent string shows the system information as Macintosh; Intel Mac OS X 10_15_7 which is the native platform the browser is running on. AppleWebKit/537.36 is the platform used by my browser and it also shows that Chrome is the web browser I used.

• HTTP request packet's URL - `/`
The client is asking for the file at path "/" relative to the web server/host.

• Web server name and version - info.cern.ch (version not given)
This is the name of the host/web server the client is sending the HTTP GET request to.

2. Packet no. 210:



• HTTP packet type - Response
The web server (info.cern.ch) makes an HTTP response to the client (my browser), providing it with the resource it requested i.e. the HTML text data to be displayed on the page.

• HTTP response code - 200
Response code 200 means that the HTTP request has succeeded.

• HTTP response description - The HTTP response 200 OK means that the HTTP request has succeeded. Since a GET request was sent by the client, 200 OK means that the resource requested by the client (here, HTML data) is successfully fetched and has been sent in the message body (Line-based text data). The actual HTTP response in this packet is:

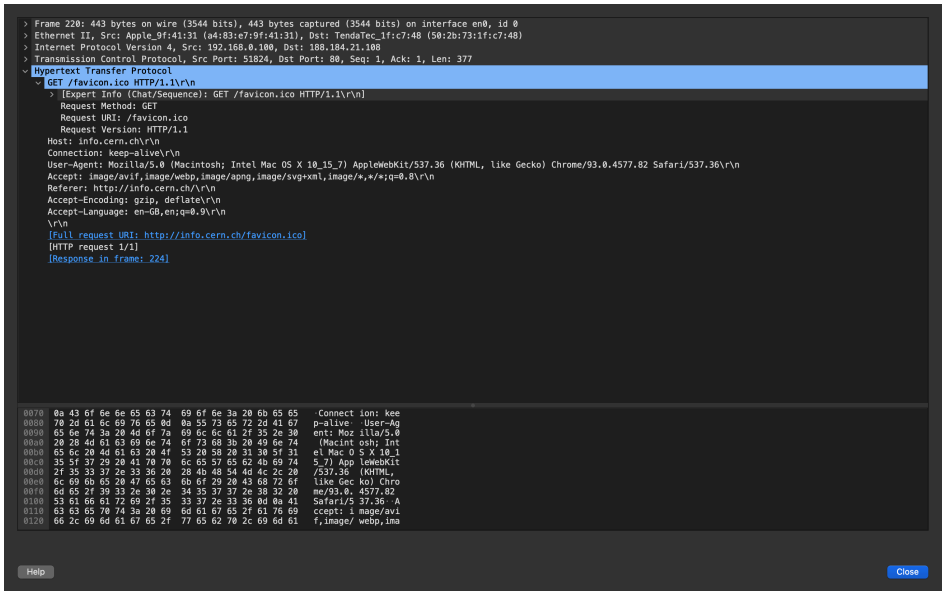
Line-based text data: text/html (13 lines)

```
    <html><head></head><body><header>\n
    <title>http://info.cern.ch</title>\n
    </header>\n
    \n
    <h1>http://info.cern.ch - home of the first website</h1>\n
    <p>From here you can:</p>\n
    <ul>\n
    <li><a href="http://info.cern.ch/hypertext/WWW/
TheProject.html">Browse the first website</a></li>\n
    <li><a href="http://line-mode.cern.ch/www/hypertext/WWW/
TheProject.html">Browse the first website using the line-mode
browser simulator</a></li>\n
    <li><a href="http://home.web.cern.ch/topics/birth-web">Learn
about the birth of the web</a></li>\n
    <li><a href="http://home.web.cern.ch/about">Learn about CERN,
the physics laboratory where the web was born</a></li>\n
    </ul>\n
```

```
</body></html>\n
```

• Web server name and version - apache (version not given)

3. Packet no. 220 (as per the screenshot)



• HTTP packet type - `Request`
Here, my web browser (client) has sent an HTTP request to initiate an action on the server.

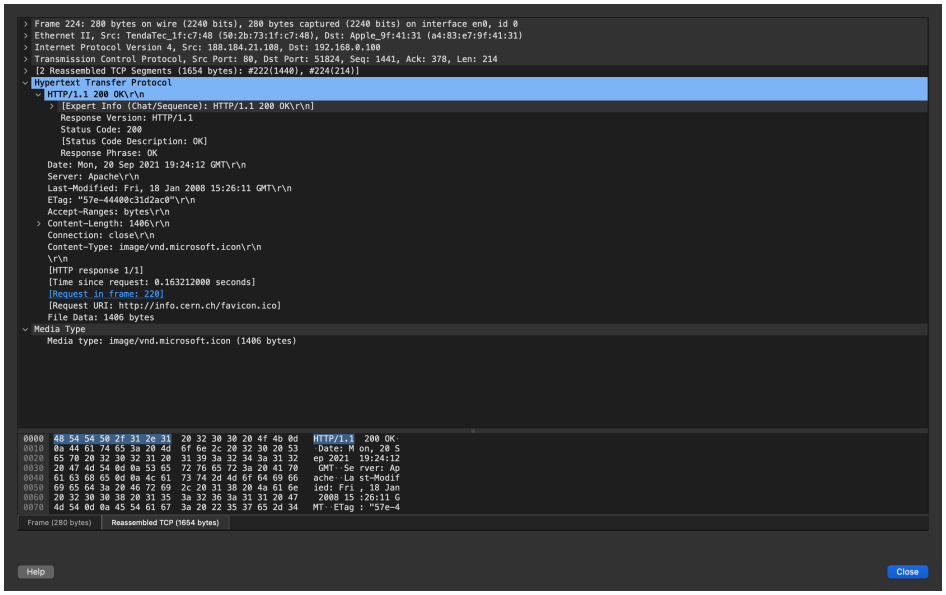
• HTTP request type - `GET`
The web browser (client) sent a HTTP GET request to the web server to request the Favicon (favicon.ico) for the web page.

• User agent type - `Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/93.0.4577.82 Safari/537.36`
I used Google Chrome to retrieve the given web page and I am working on an Apple Laptop. The user agent string shows the system information as Macintosh; Intel Mac OS X 10_15_7 which is the native platform the browser is running on. AppleWebKit/537.36 is the platform used by my browser and it also shows that Chrome is the web browser I used.

• HTTP request packet's URL - `/favicon.ico`
The client is asking for the file at path "/favicon.ico" relative to the web server/host.

• Web server name and version - info.cern.ch (version not given)
This is the name of the host/web server the client is sending the HTTP GET request to.

4. Packet no. 224 (as per the screenshot):



• HTTP packet type - Response
The web server (info.cern.ch) makes an HTTP response to the client (my browser), providing it with the resource it requested i.e. the favicon (image) to be displayed for the web page.

• HTTP response code - 200
Response code 200 means that the HTTP request has succeeded.

• HTTP response description - OK
The HTTP response 200 OK means that the HTTP request has succeeded. Since a GET request was sent by the client, 200 OK means that the resource requested by the client (here, an image/favicon) is successfully fetched and has been sent in the message body (the requested media i.e. image). The actual HTTP response in this packet is an image media file (not attached).

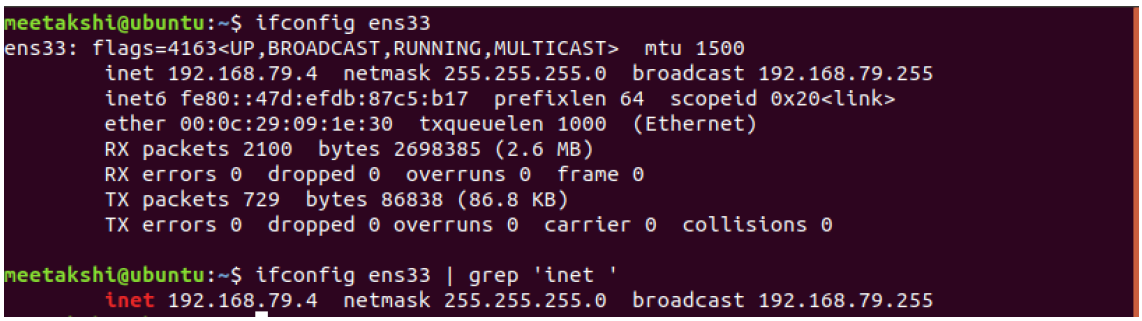
• Web server name and version - apache (version not given)

Q3.

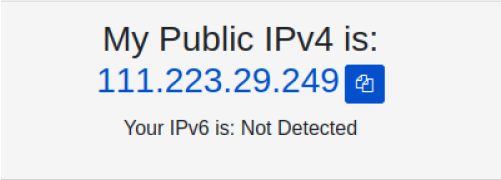
System: Ubuntu 18.04 virtual machine
Network: WiFi
Protocol: IPv4

a)
The IP address of my network interface as visible using the ifconfig command is 192.168.79.4
I am using WiFi to connect to the internet and the IPv4 protocol. That's why, I displayed the inet address. While using ifconfig, I found that the only network

interfaces visible to me were called ‘ens33’ and ‘lo’ (loopback interface). I displayed the status of the ‘ens33’ interface. I also displayed only the inet address using grep. The screenshot is attached below:



b)
According to whatismyip.com, my IP address (IPv4) was displayed as 111.223.29.249.



The IP for my machine shown by ifconfig and whatismyip.com are different. This is because ifconfig shows the private IP address of my machine within the network. The IP address shown by whatismyip.com is the public IP address for my machine, as visible to those outside the network. The public IP address is basically used to access the internet and is provided by the ISP. The private IP address, on the other hand, is in my case an address of the form 192.168.x.x which cannot be tracked by whatismyip.com (or any other external host) and is used within my network only. This is also known as Network Address Translation (NAT). My entire home network (i.e. all devices connected to my home wifi) is visible as only a single public IP address to the outside world. The incoming traffic is routed to the appropriate device via the private or local IP addresses.

Q4.

System: Ubuntu 18.04 virtual machine
Network: WiFi
Protocol: IPv4

a)
The commands I used to test whether a single packet with MTU 3000 can be sent to www.iiitd.ac.in are:

```
sudo ifconfig ens33 mtu 3000 up
ping -c 1 -M do -s 2972 www.google.com
```

In the first command, I used ifconfig to change the MTU of the ens33 interface from default 1500 to 3000.

In the second command, I used ping to send the data packet. The command line option -c specifies the number of packets to be sent. I have used '-c 1' so only a single packet will be sent www.google.com. The command line option '-M do' is used to prohibit packet fragmentation even though the packet may be of large size. This is again ensures that our packet gets sent in its entirety as one single packet instead of being transmitted in fragments. The command line option '-s 2972' specifies that the number of bytes to be sent is 2972. I chose 2972 because it is the maximum data size for an MTU of 3000 since the IPv4 header that will be of 20 bytes and ICMP header will be of 8 bytes which will make the data packet of 3000 bytes (which is the MTU size).

The test failed. Ping showed 100% packet loss. (I also tried to ping www.iiit.ac.in and www.iiitd.ac.in but got the same result). The screenshot is attached below:

```
meetakshi@ubuntu:~$ sudo ifconfig ens33 mtu 3000 up
meetakshi@ubuntu:~$ ping -c 1 -M do -s 2972 www.google.com
PING www.google.com (142.250.206.164) 2972(3000) bytes of data.

--- www.google.com ping statistics ---
1 packets transmitted, 0 received, 100% packet loss, time 0ms

meetakshi@ubuntu:~$ ping -c 1 -M do -s 2972 www.iiit.ac.in
PING www.iiit.ac.in (196.12.53.50) 2972(3000) bytes of data.

--- www.iiit.ac.in ping statistics ---
1 packets transmitted, 0 received, 100% packet loss, time 0ms

meetakshi@ubuntu:~$ ping -c 1 -M do -s 2972 www.iiitd.ac.in
PING iiitd.ac.in (103.25.231.30) 2972(3000) bytes of data.

--- iiitd.ac.in ping statistics ---
1 packets transmitted, 0 received, 100% packet loss, time 0ms
```

Some of the reasons that it didn't work are:

- The standard size MTU for Ethernet is 1500 bytes only. It is theoretically the maximum amount of data that can be transmitted by the physical link. The MTU of any higher-level protocols must fit within this MTU.
- The largest MTU supported by my router is 1500. Thus, the maximum size of a data packet that can be sent over my connection is 1500 bytes only. Since fragmenting the packet is not allowed, this very large packet of size 3000 bytes experiences packet loss.
- The largest MTU for receiver's router could also be less than 3000 bytes which would again, limit the size of the packet that can be sent over that connection for similar reasons as mentioned above.
- The same limitation as in the first and second point could apply to the intermediate nodes.
- There could have been encapsulation of additional protocols (like IPsec) that could have increased the overall frame size.

b)

The command to display all active TCP connections along with their process ids is:

```
sudo netstat -at -p
```

Here, the command line option -a for netstat is used to show all currently active connections (TCP and UDP), and using -at will show only the active TCP

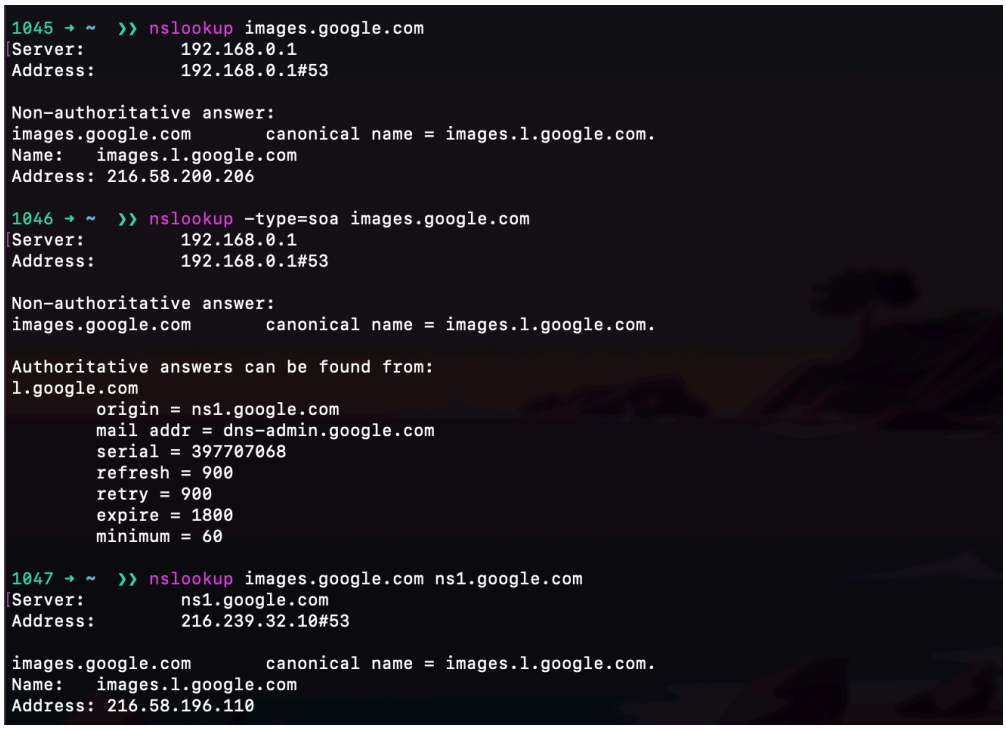
connections. By adding the -p option, the process ID/name related to the connections can be shown.

Q5.

a) *System*: MacOS Big Sur
I used nslookup to get an authoritative result for images.google.com. Simply typing `nslookup images.google.com` gives a non-authoritative answer. I obtained an authoritative answer by following these steps:

- First, I used the option -type=soa (start of authority) and nslookup responded with the name of the authoritative name server for images.google.com. The command I used was: `nslookup -type=soa images.google.com`
- After getting the name of the authoritative DNS name server (ns1.google.com), I ran the nslookup command again against the obtained name server like so- `nslookup images.google.com ns1.google.com.`

This finally gave me an authoritative answer for images.google.com. The screenshot is attached below.

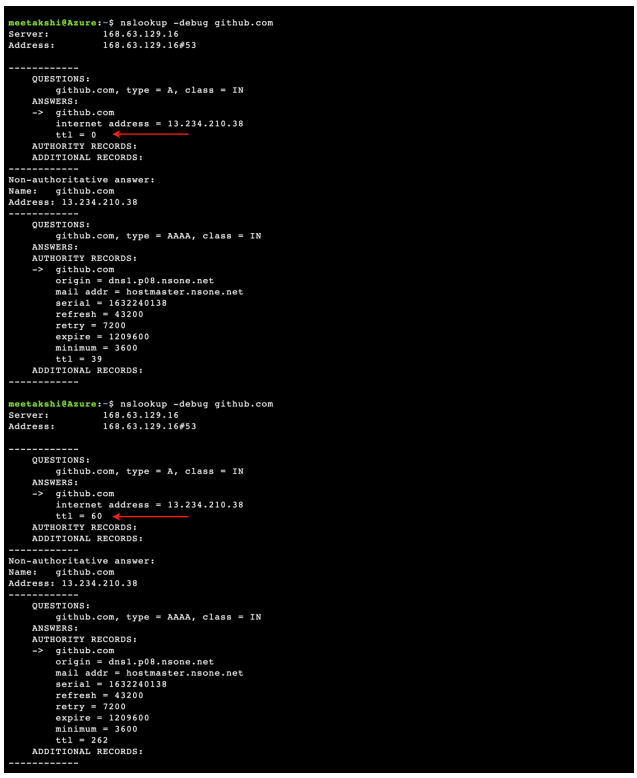


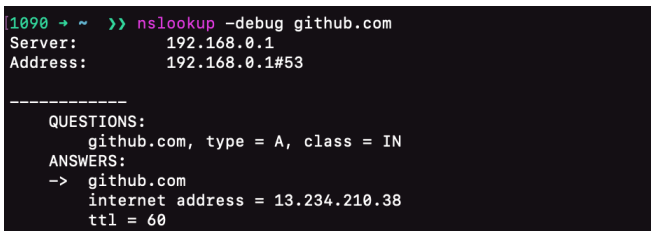
The IP addresses returned in case of a non-authoritative answer was 216.58.200.206 which differs from that returned in case of the authoritative answer i.e. 216.58.196.110. This was because the authoritative answer comes from a name server authoritative for the domain (ns1.google.com, in our case) which contains the original zone file. The non-authoritative answer, on the other hand comes from non-authoritative name servers that do not have the original zone file. images.google.com has different IPs probably for load balancing.

b) *System*: macOS Big Sur and Azure Cloud Shell (bash)

I also used the nslookup command on macOS and azure cloud shell to find the ttl for github.com like so `nslookup -debug github.com`.

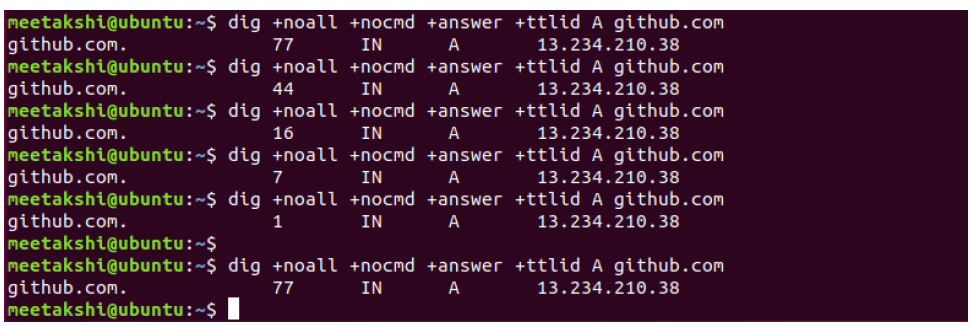
Initially, the ttl on the local DNS I obtained using the above command was 60 seconds for github.com on both macOS terminal and Azure cloud shell. On subsequent commands, the ttl reduced by the seconds until it reached 0 and became 60 seconds all over again. The screenshots showing the ttl on local DNS for github.com on Azure cloud shell (marked by red arrows) and macOS respectively is given below:





Time to live is the period of time or hops that a packet can exist in a network before being discarded in other words, ttl specifies the maximum amount of time a DNS server might cache the particular DNS record for. Since the local DNS ttl for github.com shown by the mentioned command is 60 seconds, the entry lives on the local DNS for 60 seconds after which it expires as seen in the first screenshot. Hence, the entry will expire after 60 seconds.

I also used dig command on linux (Virtual Machine) to find the time to live (ttl) of github.com on the local DNS like so:
`dig +noall +nocmd +answer +ttlid A github.com`

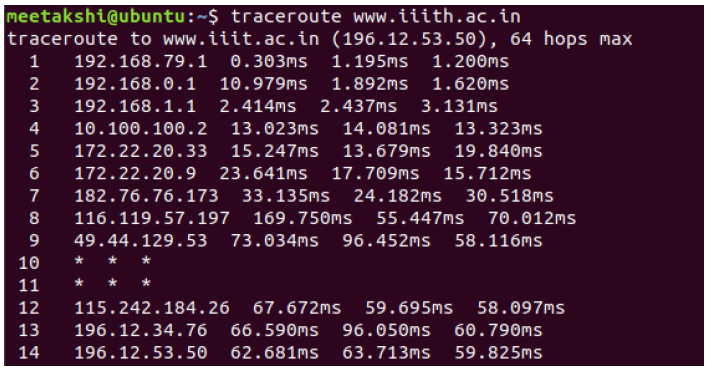


On this machine, the ttl on local DNS for github.com is 77 seconds and reduces on subsequent executions of the commands as expected. The entry here expires after 77 seconds.

Q6.

System: Ubuntu 18.04 virtual machine
Network: WiFi
Protocol: IPv4

a)
The screenshot after running `traceroute www.iiith.ac.in` is given below:

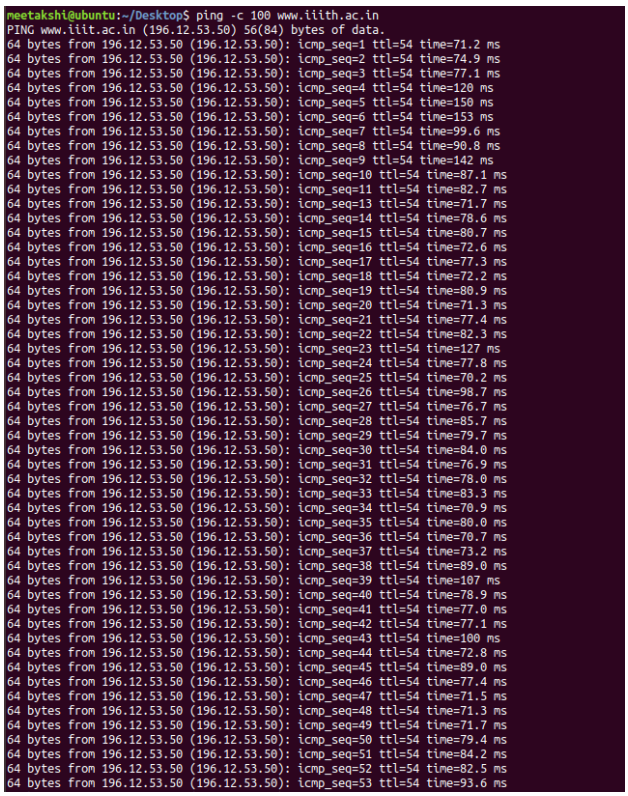


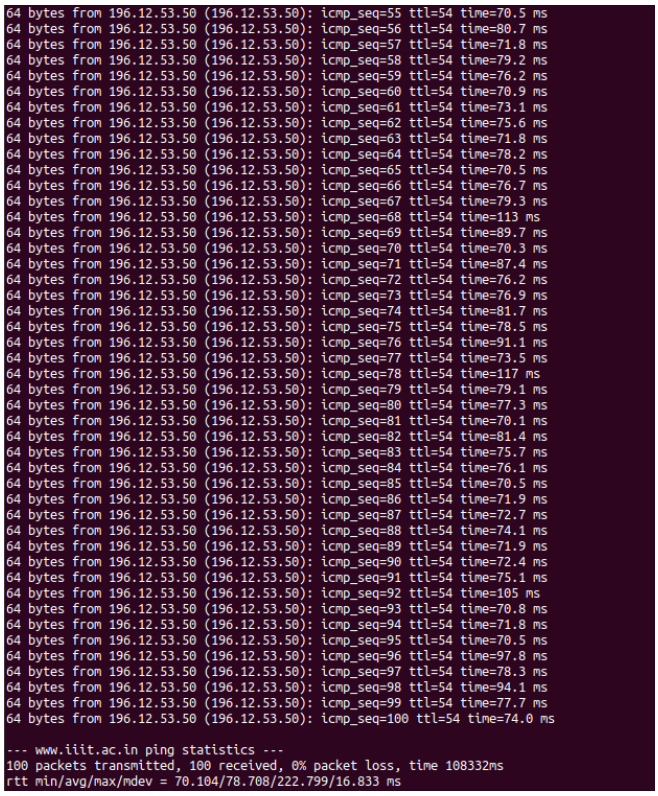
Not considering the ones with the "* * *", there are 12 intermediate hosts visible. The average latency to each host can be found by averaging the displayed RTT for the three data packets sent by traceroute. The IP address and average latency to each hop are:

Hop	IP Address	Average Latency (ms)
1	192.168.79.1	0.899
2	192.168.0.1	4.830
3	192.168.1.1	2.661
4	10.100.100.2	13.476
5	172.22.20.33	16.255
6	172.22.20.9	14.192
7	182.76.76.173	29.278
8	116.119.57.197	98.403
9	49.44.129.53	75.867
12	115.242.184.26	61.821
13	196.12.34.76	74.477
14	196.12.53.50	62.073

b)
I sent 100 pings to www.iiith.ac.in using the command- `ping -c 100 www.iiith.ac.in`

The average latency (average RTT) shown by ping is 78.708 ms. The screenshot is attached below:





c)
Adding the ping latency of all intermediate hosts from a) gives 305.778 ms. This value is considerably higher than the average ping latency obtained from b) i.e. 78.708 ms and they do not match.
This is because Ping basically sends an ICMP (echo request) packet to the destination address and waits for a reply. The packet is forwarded until it reaches the destination from where it is again sent back and received at the source host. This RTT is what is noted by the Ping command. Traceroute, however, (here) sends 3 UDP datagrams to each intermediate host with increasing TTL and when the host transmits the ICMP message “TTL Expired” for the packet, the RTT is displayed. Hence, each average RTT value for an intermediate host is actually the time taken for that packet to go from source host (my machine) to the intermediate host (router, for example). Adding all these times gives the sum of RTT to reach each intermediate host as opposed to the RTT for just reaching the destination host (like what ping does). That’s why the latency values from b) i.e. 78.708 ms and 305.778 ms are different.

d)
The maximum of average ping latency amongst the intermediate hosts is 98.403 ms for host with IP 116.119.57.197. The average latencies may be comparable, but

qualitatively, these two things are not equal (average ping latency obtained from b) is 78.708 ms)
98.403 ms is the time to reach the intermediate host 116.119.57.197, not the final destination address. On the public internet, the routes are not static. Factors like load balancing, traffic etc. causes the intermediate routers to change the routing rules as and when necessary. So, the route taken by the UDP datagram to reach the intermediate host 116.119.57.197 may not at all be the route taken by the UDP datagram to reach www.iiith.ac.in. Similarly, the route taken by the ICMP packets of the ping command may be different too. Note that we are actually trying to reach the intermediate nodes that form a path to the destination www.iiith.ac.in with UDP datagrams in traceroute. It is thus expected that sometimes, the route taken by a datagram for an intermediate host B may be different than that taken by a datagram for a host A that lies just before B. Also, to reiterate 98.403 ms is the time to reach the intermediate host 116.119.57.197, not the final destination address. Hence the difference in latencies.
But, it may be possible that the route Ping takes and the route the last 3 traceroute datagrams take to reach the final destination may be same and reaching all other intermediate hosts in traceroute takes lesser time than reaching the destination host. In this case, yes, the maximum of average ping latency amongst the intermediate hosts in traceroute (RTT to destination host) may be comparable to the average ping latency obtained from b).

e)
I performed the reverse DNS lookup for each intermediate host using the nsloookup command like so: `nslookup <IP Address>`
For commands where nslookup couldn't provide the host name, I used mxtoolbox (https://mxtoolbox.com/ReverseLookup.aspx) and jotted down my observation in the 'comments' column.
All valid hostnames/aliases are in the table below:

Hop	IP Address	Host Name and Alias	Comments
1	192.168.79.1	(private network)	Address range 192.168.0.0–192.168.255.255 is reserved for private IP addresses.
2	192.168.0.1	(private network)	Address range 192.168.0.0–192.168.255.255 is reserved for private IP addresses.
3	192.168.1.1	(private network)	Address range 192.168.0.0–192.168.255.255 is reserved for private IP addresses.
4	10.100.100.2	(private network)	Address range 10.0.0.0–10.255.255.255 is reserved for private IP addresses.
5	172.22.20.33	(private network)	Address range 172.16.0.0–172.31.255.255 is reserved for private IP addresses.
6	172.22.20.9	(private network)	Address range 172.16.0.0–172.31.255.255 is reserved for private IP addresses.
7	182.76.76.173	nsg-static-173.76.76.182-airtel.com	
8	116.119.57.197	-	No DNS record was found
9	49.44.129.53	-	No DNS record was found

Hop	IP Address	Host Name and Alias	Comments
12	115.242.184.26	115.242.184.26.static.jio.com.	
13	196.12.34.76	-	No DNS record was found
14	196.12.53.50	-	No DNS record was found

Q7.

System: Ubuntu 18.04 virtual machine
Network: WiFi
Protocol: IPv4

I configured the firewall settings using `iptables` - Linux's built in firewall. My default firewall configuration looked like this:

```
meetakshi@ubuntu:~$ sudo iptables -L
Chain INPUT (policy ACCEPT)
target     prot opt source               destination

Chain FORWARD (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
```

There are no rules set in any policy chain including the INPUT policy chain. The INPUT chain is for all packets meant for the host computer i.e. packets going into local sockets. It controls the behaviour of incoming connections. Since the ping to the IP address 127.0.0.1- which is the loopback or localhost- had to fail, I added a rule to the INPUT chain to drop all packets meant for 127.0.0.1 from the incoming traffic. I did this using the command- `sudo iptables -A INPUT -s 127.0.0.1 -j DROP`

The firewall configuration looked like this now, with a new rule added to the INPUT chain:

```
meetakshi@ubuntu:~$ sudo iptables -L
Chain INPUT (policy ACCEPT)
target     prot opt source               destination
DROP       all  --  localhost            anywhere

Chain FORWARD (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
```

After this, whenever I pinged the localhost using `ping -c 1 127.0.0.1`, it always shows 100% packet loss because that packet is filtered out by the firewall and dropped owing to the configuration I added. The screenshot below shows that the ping command failed for 127.0.0.1 with 100% packet loss.

```
meetakshi@ubuntu:~$ ping -c 1 127.0.0.1
PING 127.0.0.1 (127.0.0.1) 56(84) bytes of data.

--- 127.0.0.1 ping statistics ---
1 packets transmitted, 0 received, 100% packet loss, time 0ms
```