

Assignment 1 Report

Computer Networks

Members:
Soham Shrivastava (23110315)
Arin Mehta (23110038)

Github Repository: https://github.com/SohamShrivastava/CN_Assignment_1

Task 1

In this task, we implemented the DNS-Client server connection where the client reads DNS server query from reading pcap file (we used 3.pcap), extracts the domain name and attaches a custom header which contains the current timestamp and query ID in the format of “HHMMSSID”. This combined packet is then sent over a TCP connection to the server.

On the server side, the packet is received and parsed. It separates the custom header from the DNS query and then applies certain rules to generate the response IP from the IP pool of 15 IP addresses.

Then this response IP is sent back to the client. And then the client receives this response IP and stores the result in csv file.

```
o (env) (base) sohamshrivastava@Sohams-MacBook-Air-10 CN_Assignment_1 % python3 server.py
server listening on 127.0.0.1:12345
connected by ('127.0.0.1', 58808)
received query for: netflix.com., with header: 23112700
resolved ip: 192.168.1.11
response sent back to client

connected by ('127.0.0.1', 58809)
received query for: linkedin.com., with header: 23112801
resolved ip: 192.168.1.12
response sent back to client

connected by ('127.0.0.1', 58810)
received query for: example.com., with header: 23112902
resolved ip: 192.168.1.13
response sent back to client

connected by ('127.0.0.1', 58811)
received query for: google.com., with header: 23112903
resolved ip: 192.168.1.14
response sent back to client

connected by ('127.0.0.1', 58812)
received query for: facebook.com., with header: 23113004
resolved ip: 192.168.1.15
response sent back to client

connected by ('127.0.0.1', 58813)
received query for: amazon.com., with header: 23113005
resolved ip: 192.168.1.11
response sent back to client
```

Fig : output from server.py

```
(env) (base) sohamshrivastava@Sohams-MacBook-Air-10:~/CN_Assignment_1% python3 client.py
Starting to process packets:
```

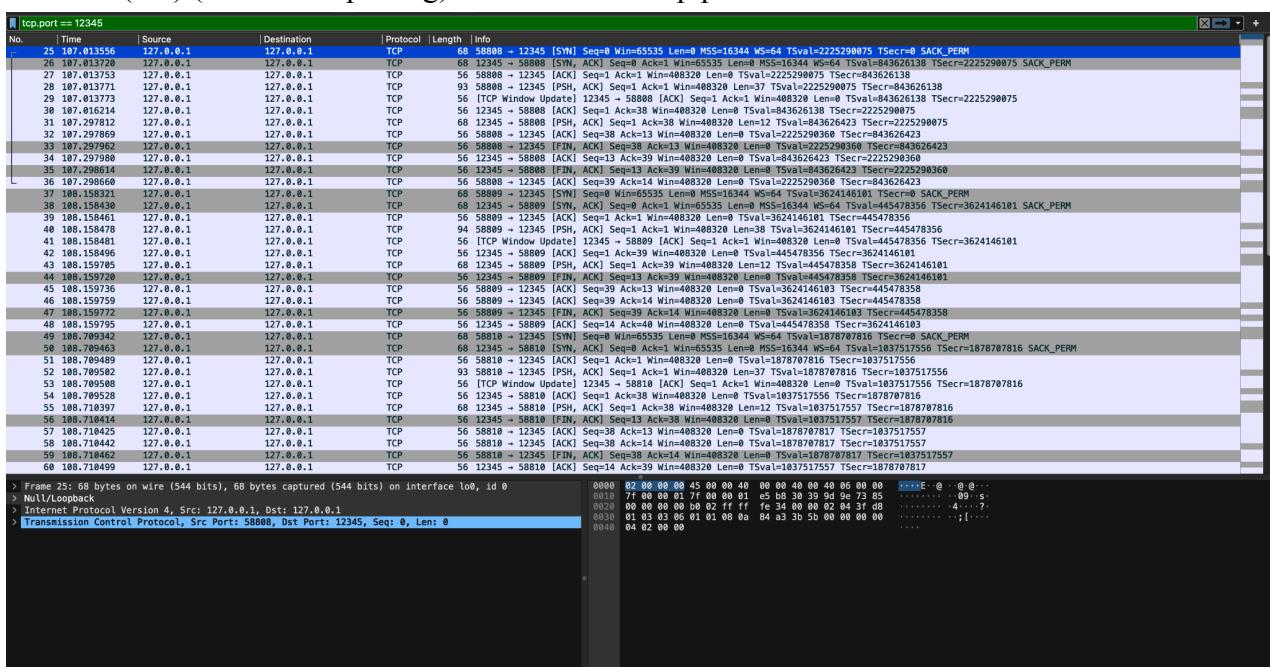
```
Query ID: 0, Domain: netflix.com., Resolved IP: 192.168.1.11
Query ID: 1, Domain: linkedin.com., Resolved IP: 192.168.1.12
Query ID: 2, Domain: example.com., Resolved IP: 192.168.1.13
Query ID: 3, Domain: google.com., Resolved IP: 192.168.1.14
Query ID: 4, Domain: facebook.com., Resolved IP: 192.168.1.15
Query ID: 5, Domain: amazon.com., Resolved IP: 192.168.1.11
summary of queries:
```

	Custom Header value (HHMMSSID)	Domain name	Resolved IP address
0	23112700	netflix.com.	192.168.1.11
1	23112801	linkedin.com.	192.168.1.12
2	23112902	example.com.	192.168.1.13
3	23112903	google.com.	192.168.1.14
4	23113004	facebook.com.	192.168.1.15
5	23113005	amazon.com.	192.168.1.11

report saved as 'dns_report.csv'

Fig: output from client.py

To analyse the background workflow, we analysed the packets using Wireshark on the loopback interface (lo0) (for local capturing) with the filter “tcp.port == 12345”.



First, the connection setup is done using the 3-way handshake:

The client sends a **SYN** packet to request a connection, the server replies with **SYN, ACK** to acknowledge, and finally, the client responds with **ACK** to confirm. This establishes the connection between client and server.

Now, during data transfer, **PSH** and **ACK** flags were used. The **PSH** flag indicates that the client is sending data (pushing) to the server, while **ACK** confirms receipt of previous data.

We can also see the data that is sent and received by the client and server.

Here on the right side, the highlighted text shows the Custom Header (23112700) and website name (netflix.com), which is sent by the client.

Now, after acknowledging the received data server will send the resolved IP to the client.

Using the same PSH, ACK flags, the server sends the resolved IP (here, 192.168.1.11) back to the client.

Now, after acknowledging the received IP address, our TCP connection gets closed, which can be observed using **FIN, ACK** flags.

For every response a new TCP connection has been made and get closed after the client gets the response IP address. This shows our non-persistent implementation of our TCP connection.

Custom Header value (HHMMSSID)	Domain name	Resolved IP address
23112700	<u>netflix.com.</u>	192.168.1.11
23112801	<u>linkedin.com.</u>	192.168.1.12
23112902	<u>example.com.</u>	192.168.1.13
23112903	<u>google.com.</u>	192.168.1.14
23113004	<u>facebook.com.</u>	192.168.1.15
23113005	<u>amazon.com.</u>	192.168.1.11

Fig: Final DNS report from our 3.Pcap File

Task 2

```
arinmehta@Arins-MacBook-Pro-4 ~ % traceroute www.google.com
traceroute to www.google.com (142.251.222.100), 64 hops max, 40 byte packets
 1  10.7.0.5 (10.7.0.5)  15.381 ms  3.591 ms  3.033 ms
 2  172.16.4.7 (172.16.4.7)  3.544 ms  3.379 ms  3.583 ms
 3  14.139.98.1 (14.139.98.1)  5.134 ms  4.776 ms  5.327 ms
 4  10.117.81.253 (10.117.81.253)  3.170 ms  3.790 ms  3.147 ms
 5  10.154.8.137 (10.154.8.137)  11.866 ms  11.200 ms  11.896 ms
 6  10.255.239.170 (10.255.239.170)  12.324 ms  10.768 ms  11.439 ms
 7  10.152.7.214 (10.152.7.214)  11.695 ms  11.198 ms  11.772 ms
 8  72.14.204.62 (72.14.204.62)  11.665 ms  12.880 ms  11.617 ms
 9  * * *
10  142.250.227.74 (142.250.227.74)  28.184 ms
    192.178.86.246 (192.178.86.246)  13.738 ms
    142.250.227.74 (142.250.227.74)  22.251 ms
11  192.178.110.106 (192.178.110.106)  12.535 ms
    192.178.110.104 (192.178.110.104)  12.644 ms
    192.178.110.206 (192.178.110.206)  22.134 ms
12  pnbomb-az-in-f4.1e100.net (142.251.222.100)  12.503 ms
    142.250.209.71 (142.250.209.71)  13.711 ms
    pnbomb-az-in-f4.1e100.net (142.251.222.100)  12.261 ms
arinmehta@Arins-MacBook-Pro-4 ~ %
```

Fig.1: traceroute www.google.com on MacOS

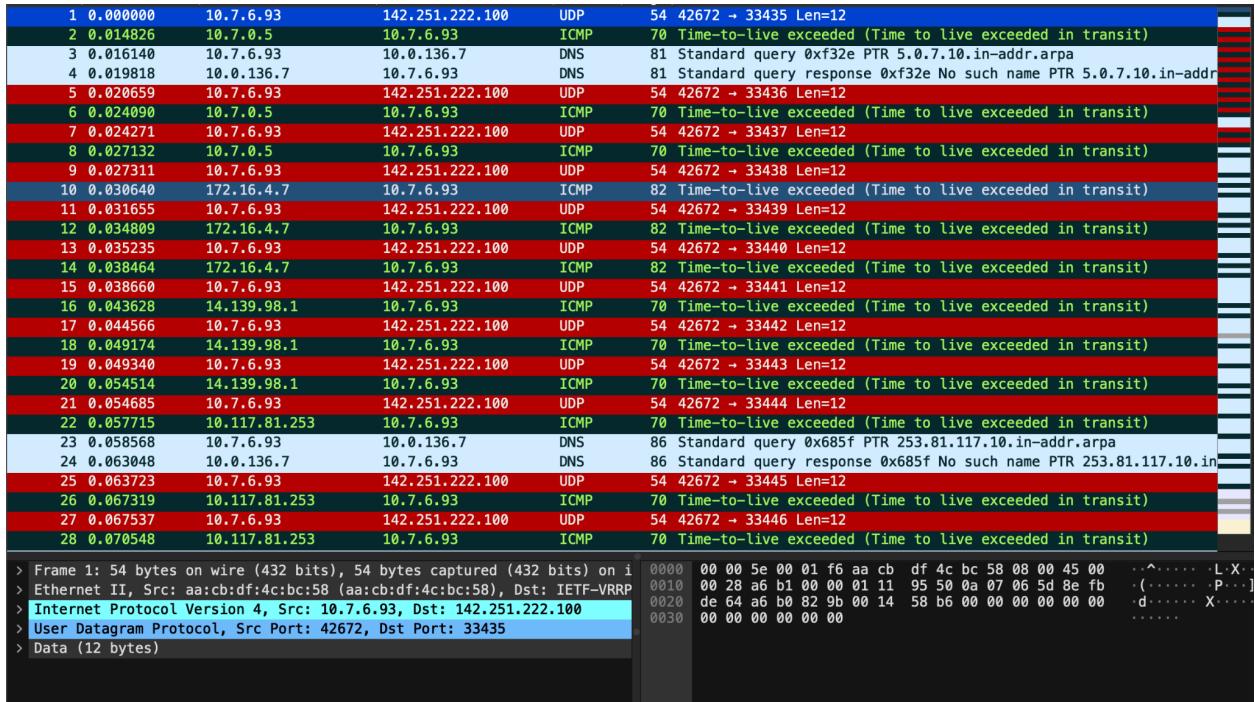


Fig.2: Wireshark screenshot for the corresponding macOS command

```
C:\Users\bhavy>tracert www.google.com

Tracing route to www.google.com [142.251.42.68]
over a maximum of 30 hops:

 1   10 ms    3 ms    3 ms  10.240.0.2
 2   12 ms    2 ms    2 ms  10.3.0.29
 3   17 ms    2 ms    2 ms  10.3.0.5
 4   14 ms    5 ms    2 ms  172.16.4.7
 5   15 ms    5 ms    5 ms  14.139.98.1
 6   13 ms    2 ms    2 ms  10.117.81.253
 7   34 ms    24 ms   22 ms  10.154.8.137
 8   49 ms    41 ms   41 ms  10.255.239.170
 9   58 ms    45 ms   45 ms  10.152.7.214
10   60 ms    55 ms   58 ms  72.14.204.62
11   71 ms    69 ms   67 ms  72.14.239.103
12   23 ms    18 ms   17 ms  142.251.69.105
13   68 ms    71 ms   73 ms  bom12s21-in-f4.1e100.net [142.251.42.68]

Trace complete.
```

Fig.3: traceroute www.google.com on Windows

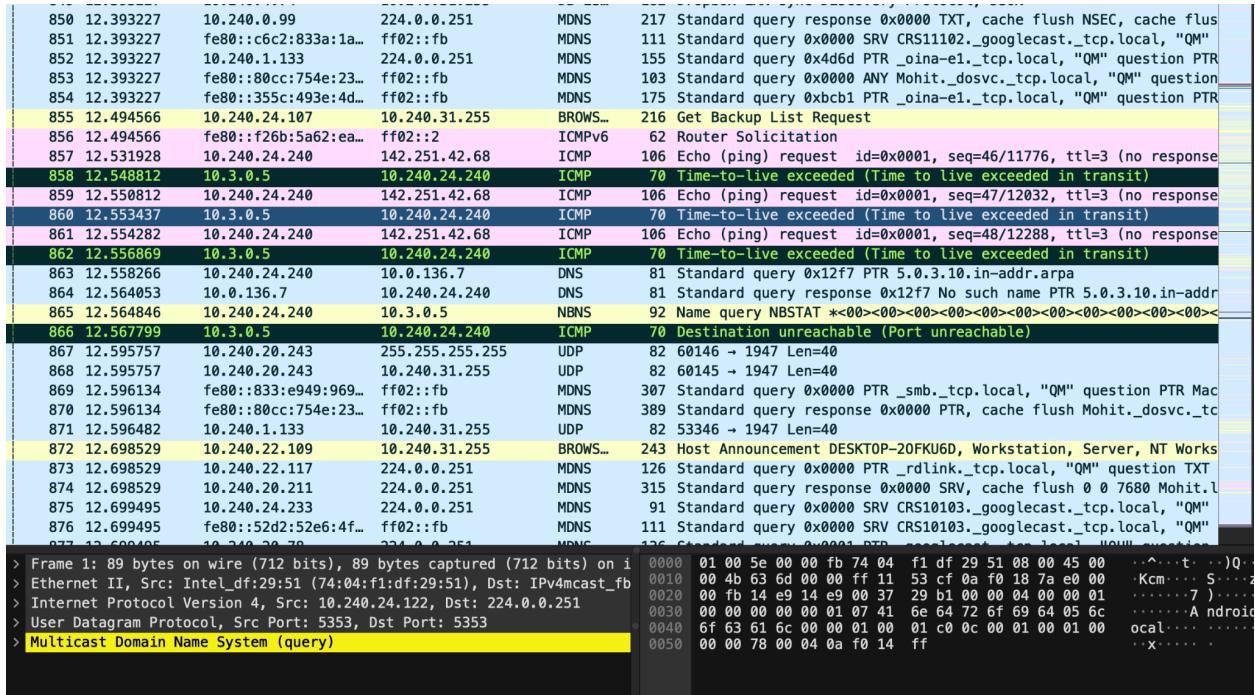


Fig.4: Wireshark screenshot for the corresponding Windows command

Question 1:

From the above screenshots, we can see that macOS(it has a close resemblance to Linux due to its Unix ancestor) uses UDP to send requests to port 33435 and above. The selected frame in the screenshot is the first frame, and the subsequent frames have the destination port as increments of 33435. The ICMP replies that we see are the TTL-exceeded replies.

Windows uses ICMP echo requests, as seen in Fig.4. We see that the replies are ICMP TTL-exceeded replies.

Thus, we can conclude that Linux traceroute uses UDP(User Datagram Protocol) by default and Windows traceroute uses ICMP echo by default.

Question 2:

In Fig.1, we can see that the 9th hop has output ***. Reasons why the router might not reply are:

1. Packets may be dropped by the firewall.
2. The destination server does not respond, either due to the service being unavailable or because it blocks probes.
3. ICMP responses may be disabled by configuration.

Question 3:

As we can see in Fig.2, the destination port increments on each successive port. The first destination port, as seen in frame 1, is 33435.

Question 4:

For macOS, at the final hop, we can see that the ICMP response is “*Destination unreachable (Port unreachable)*”, whereas in the intermediate hops, we get the ICMP “*Time-to-live exceeded (Time to live exceeded in transit)*” response. The screenshot of the same is attached below.

No.	Time	Source	Destination	Protocol	Length	Info
63	1.594023	10.7.6.93	142.251.222.100	UDP	54	42672 → 33459 Len=12
64	6.596168	10.7.6.93	142.251.222.100	UDP	54	42672 → 33460 Len=12
65	11.601334	10.7.6.93	142.251.222.100	UDP	54	42672 → 33461 Len=12
66	13.196712	10.7.6.93	52.123.173.200	TLSv1...	97	Encrypted Alert
67	16.606449	10.7.6.93	142.251.222.100	UDP	54	42672 → 33462 Len=12
68	16.634270	142.250.227.74	10.7.6.93	ICMP	110	Time-to-live exceeded (Time to live exceeded in transit)
69	16.635944	10.7.6.93	10.0.136.7	DNS	87	Standard query 0x8dd8 PTR 74.227.250.142.in-addr.arpa
70	16.656675	10.0.136.7	10.7.6.93	DNS	147	Standard query response 0x8dd8 No such name PTR 74.227.250.142.in-addr.arpa
71	16.657631	10.7.6.93	142.251.222.100	UDP	54	42672 → 33463 Len=12
72	16.671184	192.178.86.246	10.7.6.93	ICMP	82	Time-to-live exceeded (Time to live exceeded in transit)
73	16.672022	10.7.6.93	10.0.136.7	DNS	87	Standard query 0xf5a8 PTR 246.86.178.192.in-addr.arpa
74	16.689361	10.0.136.7	10.7.6.93	DNS	147	Standard query response 0xf5a8 No such name PTR 246.86.178.192.in-addr.arpa
75	16.690000	10.7.6.93	142.251.222.100	UDP	54	42672 → 33464 Len=12
76	16.712030	142.250.227.74	10.7.6.93	ICMP	110	Time-to-live exceeded (Time to live exceeded in transit)
77	16.713338	10.7.6.93	142.251.222.100	UDP	54	42672 → 33465 Len=12
78	16.725759	192.178.110.106	10.7.6.93	ICMP	110	Time-to-live exceeded (Time to live exceeded in transit)
79	16.726749	10.7.6.93	10.0.136.7	DNS	88	Standard query 0x665b PTR 106.110.178.192.in-addr.arpa
80	16.744277	10.0.136.7	10.7.6.93	DNS	148	Standard query response 0x665b No such name PTR 106.110.178.192.in-addr.arpa
81	16.744958	10.7.6.93	142.251.222.100	UDP	54	42672 → 33466 Len=12
82	16.757488	192.178.110.104	10.7.6.93	ICMP	110	Time-to-live exceeded (Time to live exceeded in transit)
83	16.758259	10.7.6.93	10.0.136.7	DNS	88	Standard query 0xd2de PTR 104.110.178.192.in-addr.arpa
84	16.775700	10.0.136.7	10.7.6.93	DNS	148	Standard query response 0xd2de No such name PTR 104.110.178.192.in-addr.arpa
85	16.776294	10.7.6.93	142.251.222.100	UDP	54	42672 → 33467 Len=12
86	16.798261	192.178.110.206	10.7.6.93	ICMP	70	Time-to-live exceeded (Time to live exceeded in transit)
87	16.799712	10.7.6.93	10.0.136.7	DNS	88	Standard query 0x665a PTR 206.110.178.192.in-addr.arpa
88	16.815146	10.0.136.7	10.7.6.93	DNS	148	Standard query response 0x665a No such name PTR 206.110.178.192.in-addr.arpa
89	16.815640	10.7.6.93	142.251.222.100	UDP	54	42672 → 33468 Len=12
90	16.827950	142.251.222.100	10.7.6.93	ICMP	70	Destination unreachable (Port unreachable)

```
> Frame 90: 70 bytes on wire (560 bits), 70 bytes captured (560 bits) on
> Ethernet II, Src: Cisco_6c:2d:7f (88:1d:fc:6c:2d:7f), Dst: aa:cb:df:4c:
> Internet Protocol Version 4, Src: 142.251.222.100, Dst: 10.7.6.93
> Internet Control Message Protocol
0000  aa cb df 4c bc 58 88 1d fc 6c 2d 7f 08 00 45 80  ..L-X..l...
0010  00 38 00 00 00 73 01 c9 81 8e fb de 64 0a 07  .8...s...
0020  06 5d 03 03 7a e6 00 00 00 00 45 80 00 28 a6 d2  .]..z...E...
0030  00 00 01 11 94 af 0a 07 06 5d 8e fb de 64 a6 b0  .....[...]..X.
0040  82 bc 00 14 58 95  .....X.
```

Fig.5: Wireshark screenshot displaying the final hop on macOS

For windows, at the final hop, we can see that at the final hop, the response is ICMP Echo Reply, whereas in intermediate hops, we get the ICMP “*Time-to-live exceeded (Time to live exceeded in transit)*” response. The screenshot of the same is attached below.

5344	69.698148	10.240.19.194	239.255.255.250	SSDP	1/9 M-SEARCH * HTTP/1.1
5345	69.705309	142.251.42.68	10.240.24.240	ICMP	106 Echo (ping) reply id=0x0001, seq=76/19456, ttl=113 (request i
→ 5346	69.707455	10.240.24.240	142.251.42.68	ICMP	106 Echo (ping) request id=0x0001, seq=77/19712, ttl=13 (reply in 5
5347	69.726507	10.240.19.194	239.255.255.250	SSDP	179 M-SEARCH * HTTP/1.1
5348	69.735451	10.240.25.25	10.240.31.255	UDP	264 59128 → 59870 Len=222
5349	69.735451	10.240.25.231	10.240.31.255	BROWS...	216 Get Backup List Request
5350	69.735451	10.240.20.78	224.0.0.251	MDNS	184 Standard query 0x0002 PTR _googlecast._tcp.local, "QM" question
5351	69.735451	10.240.20.205	10.240.31.255	BROWS...	216 Get Backup List Request
5352	69.736180	fe80::d2ad:2c45:51...	ff02::fb	MDNS	204 Standard query 0x0002 PTR _googlecast._tcp.local, "QM" question
5353	69.736180	10.240.25.231	10.240.31.255	NBNS	92 Name query NB WORKGROUP<1b>
5354	69.736180	10.240.25.11	224.0.0.251	MDNS	411 Standard query response 0x0000 PTR Om's iPad._companion-link._tc
5355	69.736734	fe80::e7b9:2b:72bc...	ff02::fb	MDNS	328 Standard query response 0x0000 PTR CRS10202._googlecast._tcp.loc
5356	69.736734	10.240.1.12	224.0.0.251	MDNS	104 Standard query 0x0000 TXT 88038F28B75B@CRS10201._raop._tcp.local
← 5357	69.779013	142.251.42.68	10.240.24.240	ICMP	106 Echo (ping) reply id=0x0001, seq=77/19712, ttl=13 (request i
5358	69.781834	10.240.24.240	142.251.42.68	ICMP	106 Echo (ping) request id=0x0001, seq=78/19968, ttl=13 (reply in 5
5359	69.802433	10.240.25.25	239.255.255.250	UDP/X...	698 64287 → 3702 Len=656
5360	69.838482	10.240.22.63	224.0.0.251	MDNS	1129 Standard query response 0x0000 TXT, cache flush PTR _airplay._tc
5361	69.838482	fe80::1801:2d8c:db...	ff02::fb	MDNS	453 Standard query response 0x0000 PTR Bishal's iPad (2)._companion-
5362	69.838482	fe80::469:ab1:5737...	ff02::fb	MDNS	124 Standard query 0x0000 TXT 88038F28B75B@CRS10201._raop._tcp.local
5363	69.839465	fe80::1c68:c94c:ca...	ff02::fb	MDNS	1149 Standard query response 0x0000 TXT, cache flush PTR _airplay._tc
5364	69.839851	10.240.21.70	224.0.0.251	MDNS	678 Standard query response 0x0000 PTR 10.0.117.184 @ Himanshu's Mac
5365	69.840720	10.240.12.3	224.0.0.251	MDNS	762 Standard query response 0x0000 PTR Shubham's MacBook Pro._compan
5366	69.840903	fe80::59:5089:e658...	ff02::fb	MDNS	698 Standard query response 0x0000 PTR 10.0.117.184 @ Himanshu's Mac
5367	69.840903	fe80::425:930d:dee...	ff02::1:ff8a:2935	ICMPv6	86 Neighbor Solicitation for fe80::1c1f:c663:478a:2935 from de:36:8
5368	69.841981	fe80::cff:d4e3:ea8...	ff02::fb	MDNS	782 Standard query response 0x0000 PTR Shubham's MacBook Pro._compan
5369	69.841981	fe80::14bb:6ae5:81...	ff02::fb	MDNS	431 Standard query response 0x0000 PTR Om's iPad._companion-link._tc
5370	69.855189	142.251.42.68	10.240.24.240	ICMP	106 Echo (ping) reply id=0x0001, seq=78/19968, ttl=13 (request i
5371	69.918654	10.240.19.194	239.255.255.250	SSDP	212 M-SEARCH * HTTP/1.1

```
> Frame 5357: 106 bytes on wire (848 bits), 106 bytes captured (848 bits)
> Ethernet II, Src: Cisco_bd:55:f3 (6c:03:b5:bd:55:f3), Dst: AzureWaveTec
> Internet Protocol Version 4, Src: 142.251.42.68, Dst: 10.240.24.240
└ Internet Control Message Protocol
    Type: 0 (Echo (ping) reply)
    Code: 0
    Checksum: 0xffff [correct]
    [Frame details] [Followed] [Details]
```

0000 a8 e2 91 95 93 90 6c 03 b5 bd 55 f3 08 00 45 60 . . . l . U
0010 00 5c 00 00 00 00 71 01 6c 22 8e fb 2a 44 0a f0 . \ . q l' . *D
0020 18 f0 00 00 ff b1 00 01 00 4d 00 00 00 00 00 00
0030 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0040 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0050 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0060 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Fig.6: Wireshark screenshot displaying the final hop on Windows

Question 5:

If the firewall blocks UDP but allows ICMP the linux traceroute will not work as it uses UDP.
On the other hand, windows traceroute will continue working as it is because it uses ICMP echo.