Assignment-1

CS 331: Computer Networks

Shardul Junagade (23110297)
Rishabh Jogani (23110276)

Repository: cn-assignment-1

September 15, 2025

Contents

Repository: `https://github.com/ShardulJunagade/cn-assignment-1`

Task 1: Custom DNS Resolver

1.1 Aim

The aim of this task was to implement a custom DNS resolver system. The objective was to parse DNS query packets from a given PCAP file, add a custom header, send them to a server for resolution based on predefined rules, and log the resolved IP addresses for reporting. This task helped us understand packet parsing, custom protocol design, and time-based routing logic.

1.2 Introduction

In this assignment, we developed a client-server system to process DNS queries captured in a PCAP file. The client parsed DNS query packets, prepended a custom header containing a timestamp and sequence ID, and sent these packets to a custom server. The server extracted the header, applied time-based routing rules to select an IP address, and responded with the resolved result. We implemented the solution using two different packet processing libraries: Scapy and dpkt, to demonstrate versatility and robustness.

The custom header format was `HHMMSSID`, where `HH` is the hour, `MM` is the minute, `SS` is the second, and `ID` is the sequence number of the DNS query. The server used this header to determine the time slot and select an IP address from a pool, as specified in a JSON rules file.

At first, we built both the client and server using the Scapy library. While Scapy was convenient for quickly inspecting packets and building DNS messages, it turned out to be too slow when processing larger PCAP files. The main reasons were: (1) Scapy decodes every field of every packet into Python objects, which is computationally heavy, (2) it stores extra metadata even when we only need the DNS question name, and (3) iterating through hundreds of thousands of packets became unreasonably slow. Because of this, we reimplemented the solution in dpkt, a much lighter and faster library. With dpkt, packet parsing became significantly quicker and better suited for large-scale PCAP analysis.

1.3 Procedure

1.3.1 PCAP Parsing and DNS Query Extraction

We first parsed the provided PCAP file to extract DNS query packets. For this, we implemented two separate clients:

- **Scapy-based client (`client_scapy.py`):** We used `rdpcap` to read the PCAP and filtered standard DNS queries (`qr == 0`). For each query, we extracted the domain via `DNSQR.qname`, skipped mDNS/service discovery (`.local.` and names starting with _), built the 8-byte header, and sent a UDP payload containing **header + domain string** to the server.

```
def build_header(seq_id: int) -> str:
    return f"{datetime.now().strftime('%H%M%S')}{seq_id:02d}"

packets = rdpcap('3.pcap')
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
seq_id = 0
```

```
for pkt in packets:
    if pkt.haslayer(DNS) and pkt[DNS].qr == 0:
        qname = pkt[DNSQR].qname.decode()
        if qname.endswith('.local.') or qname.startswith('_'):
            continue
        header = build_header(seq_id) # HHMMSSID
        message = header.encode() + qname.encode()
        s.sendto(message, ("127.0.0.1", 9999))
        seq_id += 1
```

Listing 1.1: Filtering DNS queries and sending header+domain

- **dpkt-based client (`client_dpkt.py`):** Used dpkt's PCAP and DNS parsing utilities:

```
with open(pcap_path, "rb") as f:
    pcap = dpkt.pcap.Reader(f)
    for ts, buf in pcap:
        eth = dpkt.ethernet.Ethernet(buf)
        ip = eth.data
        udp = ip.data
        if isinstance(udp, dpkt.udp.UDP) and udp.dport == 53:
            dns = dpkt.dns.DNS(udp.data)
            ...
```

1.3.2 Custom Header Construction

For each DNS query, we constructed an 8-byte custom header in the format `HHMMSSID`, where:

- `HH`: Hour (24-hour format)
- `MM`: Minute
- `SS`: Second
- `ID`: Sequence number of the DNS query (starting from 00)

This header was prepended to the outbound payload. In the Scapy client, the payload was **header + domain string** (not raw DNS bytes):

```
hhmmss = datetime.now().strftime("%H%M%S")
seq_id_str = f"{seq_id:02d}"
header = hhmmss + seq_id_str
payload = header.encode('ascii') + qname.encode()
```

1.3.3 Client-Server Communication

The client sent the modified packet (header + DNS query) to the server over UDP. The server was implemented in two variants:

- **Scapy-based server (`server_scapy.py`):** Used Scapy to parse incoming DNS queries and extract the domain name.
- **dpkt-based server (`server_dpkt.py`):** Used dpkt for the same purpose.

We configured structured logging on the dpkt server using a shared logger module before starting the UDP loop.

Example server code (dpkt with logging):

```
setup_logging("server_dpkt")

def handle_packet(data: bytes, client_addr, sock: socket.socket, rules):
header = data[:8].decode('ascii', errors='ignore')
dns_bytes = data[8:]
    dns = dpkt.dns.DNS(dns_bytes)
    domain = dns.qd[0].name.decode() if isinstance(dns.qd[0].name, bytes) else dns.qd[0].name
    domain = domain.rstrip('.')
    resolved_ip = resolve_ip_address(rules, header)
    response = f"{header}|{domain}|{resolved_ip}"
    sock.sendto(response.encode('utf-8'), client_addr)
    logging.info(f"Responded to {client_addr}: {response}")
```

Listing 1.2: dpkt server: logging + DNS parsing

1.3.4 Server-side IP Resolution

Upon receiving a packet, the server extracted the custom header and used the hour and sequence ID to select an IP address from a pool, according to time-based rules defined in a JSON file (`rules.json`). The rules specified different IP pool ranges and hash functions for morning, afternoon, and night slots. The logic was as follows:

```
hour = int(header[0:2])
session_id = int(header[6:8])
if 4 <= hour <= 11:
    slot_cfg = time_rules.get("morning")
elif 12 <= hour <= 19:
    slot_cfg = time_rules.get("afternoon")
else:
    slot_cfg = time_rules.get("night")
index = ip_pool_start + (session_id % hash_mod)
```

The rules and pool configuration were as described in the assignment and the provided rules documentation.

1.3.5 Running the dpkt server (`run_server`)

The server was started by calling `run_server(host, port, rules_path)`. It loaded the JSON rules, bound a UDP socket, and entered a loop to receive packets and dispatch them to `handle_packet`.

```
def run_server(host: str, port: int, rules_path: str):
    rules = load_rules_from_json(rules_path)
    logging.info(f"Loaded rules from {rules_path}")
    logging.info(f"IP Pool Length: {len(rules.get('ip_pool', DEFAULT_IP_POOL))}")

    server = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    server.bind((host, port))
    logging.info(f"DNS server (dpkt) listening on {host}:{port}")

    while True:
        data, addr = server.recvfrom(65535)
        handle_packet(data, addr, server, rules)
```

Listing 1.3: Server startup: run$_server$()

1.3.6 Response and Logging

The server responded with a plain text message in the format `header|domain|resolved_ip`. The client received this response and logged the results in a CSV file for reporting. Both implementations (Scapy and

dpkt) followed this protocol. All the terminal outputs generated during execution were also saved as log files in the repository: `https://github.com/ShardulJunagade/cn-assignment-1/tree/main/logs`.

1.4 Results

The system successfully parsed DNS queries from the PCAP file, sent them to the server with the custom header, and received resolved IP addresses based on the specified rules. The results were saved in CSV files (`dpkt_dns_report.csv` and `scapy_dns_report.csv`), each containing the custom header, domain name, and resolved IP address for every query.

- Both Scapy and dpkt implementations produced consistent results, demonstrating the correctness of the approach.
- The server correctly applied time-based routing rules and handled edge cases (e.g., out-of-bounds indices).
- The code was modular, well-commented, and robust against malformed packets.

Sample output (CSV):

Custom header value (HHMMSSID)	Domain name	Resolved IP address
20464700	netflix.com	192.168.1.11
20464701	linkedin.com	192.168.1.12
20464702	example.com	192.168.1.13
20464703	google.com	192.168.1.14
20464704	facebook.com	192.168.1.15
20464705	amazon.com	192.168.1.11

DNS Resolution Rule Example:

- If the hour is 20 (night slot), the pool start is 10, and hash mod is 5. For ID 00: $10 + (00\%5) = 10 \rightarrow$ 192.168.1.11
- For ID 01: $10 + (01\%5) = 11 \rightarrow$ 192.168.1.12
- For ID 04: $10 + (04\%5) = 14 \rightarrow$ 192.168.1.15

1.5 References

[1] Wireshark Documentation

[2] Scapy Documentation

[3] dpkt Documentation

[4] Python socket — Low-level networking interface

Task 2:Traceroute Protocol Behavior

2.1 Aim

The purpose of this task is to understand how the `traceroute` and `tracert` utilities work on different operating systems (Windows and macOS in this case), and to analyze its behavior using packet captures.

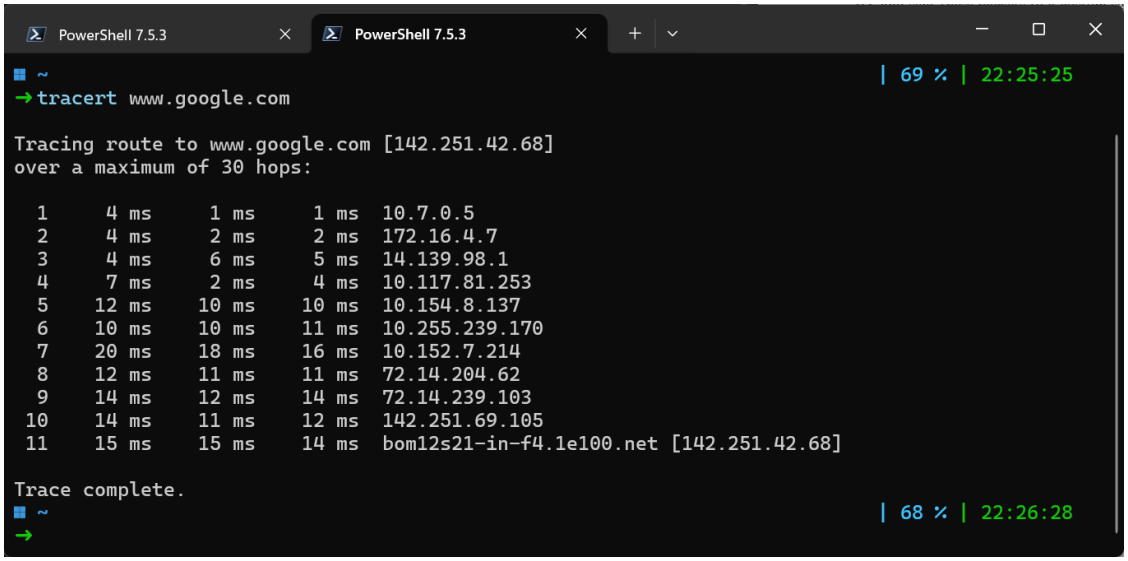
2.2 Methodology

1. On macOS, executed `traceroute www.google.com`.
2. On Windows, executed `tracert www.google.com`.
3. Captured network traffic into .pcap files using `tcpdump` (`sudo tcpdump -i en0 host www.google.com -w TracerouteMac.pcap`) for macOS and captured .pcap files using `tshark` (`tshark -i 5 -f "host www.google.com" -w TracertWin.pcap`) for Windows.
4. Opened captures in Wireshark, inspected the packets, and compared results.

2.3 Terminal Output

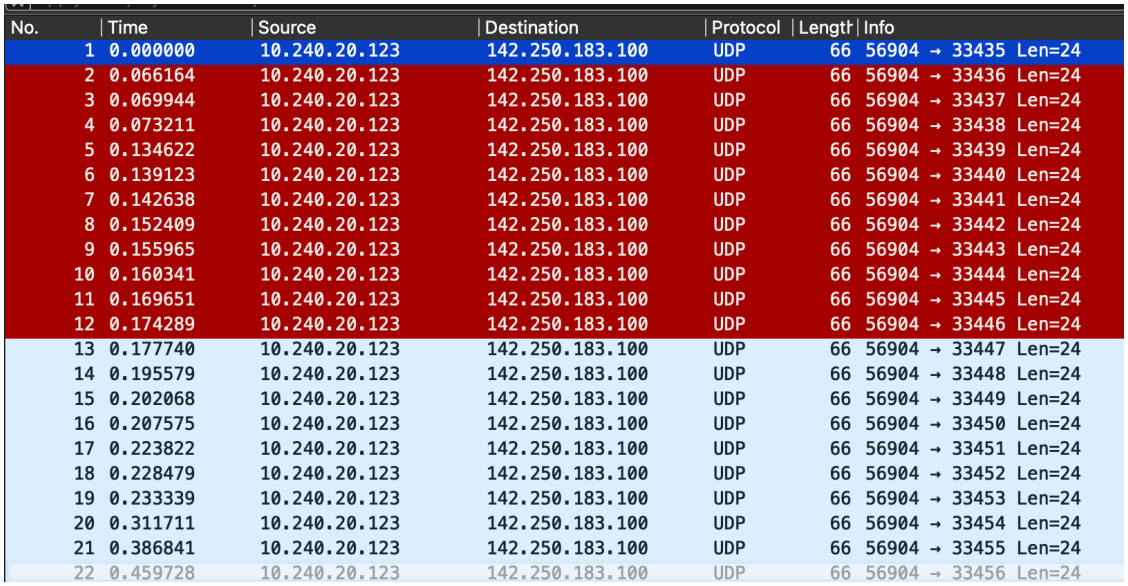
```
Last login: Sat Sep 13 11:35:04 on ttys005
rishabhsmacbook@Rishabhs-MacBook-Air ~ % traceroute www.google.com

traceroute to www.google.com (142.251.42.228), 64 hops max, 52 byte packets
 1  10.7.0.5 (10.7.0.5)  3.092 ms  2.433 ms  2.117 ms
 2  172.16.4.7 (172.16.4.7)  2.116 ms  2.498 ms  2.581 ms
 3  14.139.98.1 (14.139.98.1)  4.209 ms  4.074 ms  4.411 ms
 4  10.117.81.253 (10.117.81.253)  2.108 ms  3.202 ms  2.543 ms
 5  10.154.8.137 (10.154.8.137)  10.715 ms  10.942 ms  10.282 ms
 6  10.255.239.170 (10.255.239.170)  11.030 ms  10.540 ms  10.563 ms
 7  10.152.7.214 (10.152.7.214)  10.666 ms  10.985 ms  10.857 ms
 8  * 72.14.204.62 (72.14.204.62)  11.784 ms  71.683 ms
 9  * * *
10  pnbomb-aw-in-f4.1e100.net (142.251.42.228)  14.181 ms
    216.239.58.18 (216.239.58.18)  13.911 ms
    172.253.77.20 (172.253.77.20)  11.498 ms
```

MacOS terminal `tracert` output to www.google.com.



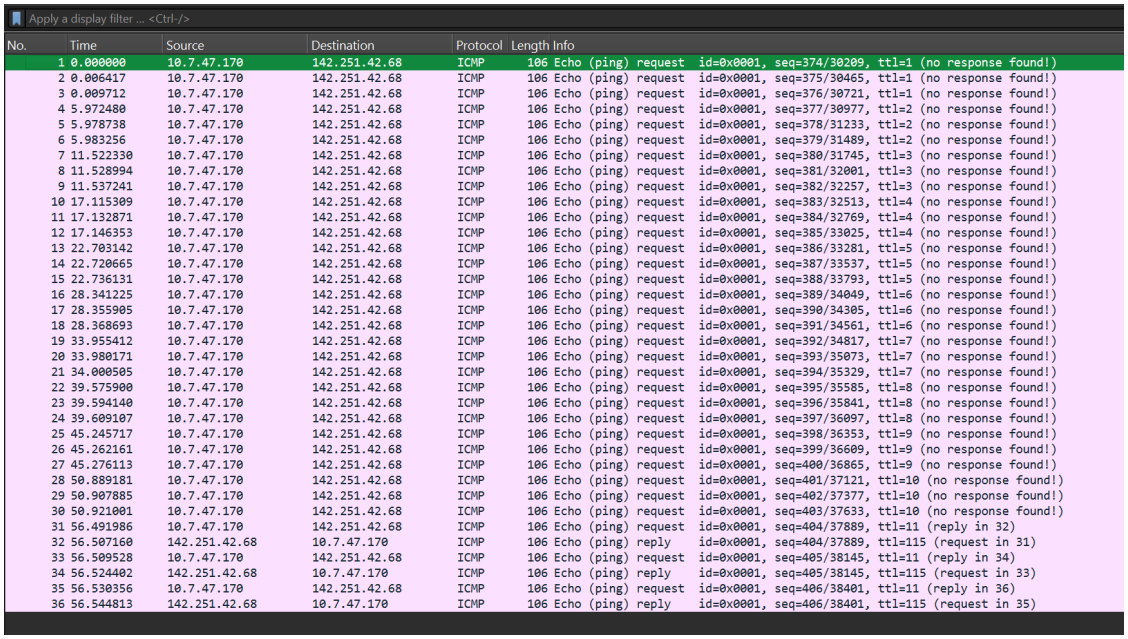
Windows terminal `tracert` output to www.google.com.

2.4 Wireshark Output



No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	10.240.20.123	142.250.183.100	UDP	66	56904 → 33435 Len=24
2	0.066164	10.240.20.123	142.250.183.100	UDP	66	56904 → 33436 Len=24
3	0.069944	10.240.20.123	142.250.183.100	UDP	66	56904 → 33437 Len=24
4	0.073211	10.240.20.123	142.250.183.100	UDP	66	56904 → 33438 Len=24
5	0.134622	10.240.20.123	142.250.183.100	UDP	66	56904 → 33439 Len=24
6	0.139123	10.240.20.123	142.250.183.100	UDP	66	56904 → 33440 Len=24
7	0.142638	10.240.20.123	142.250.183.100	UDP	66	56904 → 33441 Len=24
8	0.152409	10.240.20.123	142.250.183.100	UDP	66	56904 → 33442 Len=24
9	0.155965	10.240.20.123	142.250.183.100	UDP	66	56904 → 33443 Len=24
10	0.160341	10.240.20.123	142.250.183.100	UDP	66	56904 → 33444 Len=24
11	0.169651	10.240.20.123	142.250.183.100	UDP	66	56904 → 33445 Len=24
12	0.174289	10.240.20.123	142.250.183.100	UDP	66	56904 → 33446 Len=24
13	0.177740	10.240.20.123	142.250.183.100	UDP	66	56904 → 33447 Len=24
14	0.195579	10.240.20.123	142.250.183.100	UDP	66	56904 → 33448 Len=24
15	0.202068	10.240.20.123	142.250.183.100	UDP	66	56904 → 33449 Len=24
16	0.207575	10.240.20.123	142.250.183.100	UDP	66	56904 → 33450 Len=24
17	0.223822	10.240.20.123	142.250.183.100	UDP	66	56904 → 33451 Len=24
18	0.228479	10.240.20.123	142.250.183.100	UDP	66	56904 → 33452 Len=24
19	0.233339	10.240.20.123	142.250.183.100	UDP	66	56904 → 33453 Len=24
20	0.311711	10.240.20.123	142.250.183.100	UDP	66	56904 → 33454 Len=24
21	0.386841	10.240.20.123	142.250.183.100	UDP	66	56904 → 33455 Len=24
22	0.459728	10.240.20.123	142.250.183.100	UDP	66	56904 → 33456 Len=24

macOS `traceroute` output to www.google.com.



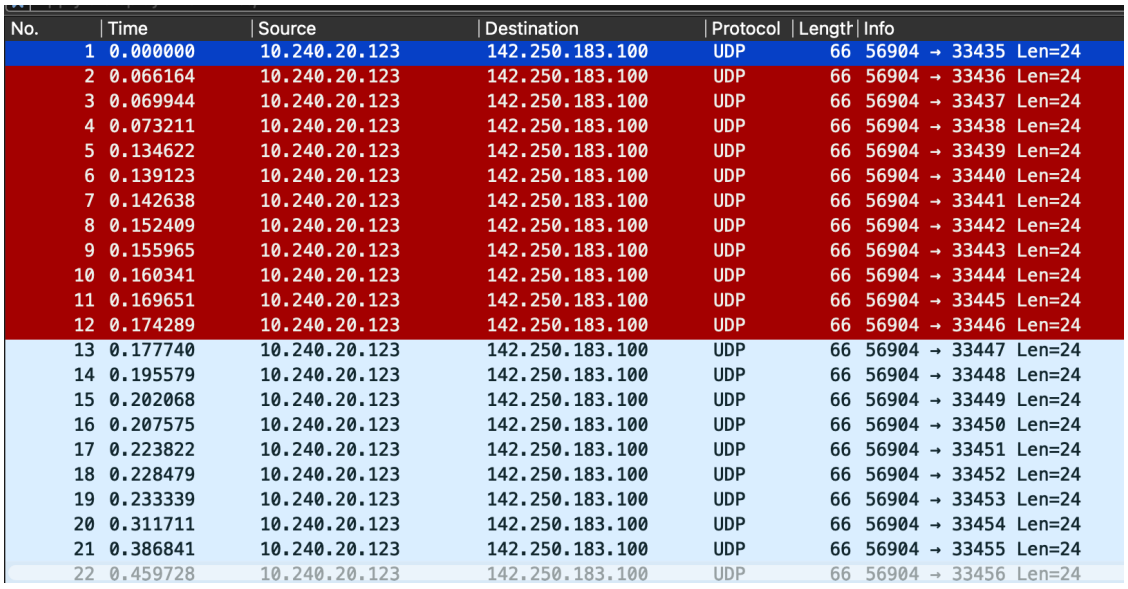
No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	10.7.47.170	142.251.42.68	ICMP	106	Echo (ping) request id=0x0001, seq=374/30209, ttl=1 (no response found!)
2	0.006417	10.7.47.170	142.251.42.68	ICMP	106	Echo (ping) request id=0x0001, seq=375/30465, ttl=1 (no response found!)
3	0.009712	10.7.47.170	142.251.42.68	ICMP	106	Echo (ping) request id=0x0001, seq=376/30721, ttl=1 (no response found!)
4	5.972480	10.7.47.170	142.251.42.68	ICMP	106	Echo (ping) request id=0x0001, seq=377/30977, ttl=2 (no response found!)
5	5.978738	10.7.47.170	142.251.42.68	ICMP	106	Echo (ping) request id=0x0001, seq=378/31233, ttl=2 (no response found!)
6	5.983256	10.7.47.170	142.251.42.68	ICMP	106	Echo (ping) request id=0x0001, seq=379/31489, ttl=2 (no response found!)
7	11.522330	10.7.47.170	142.251.42.68	ICMP	106	Echo (ping) request id=0x0001, seq=380/31745, ttl=3 (no response found!)
8	11.528994	10.7.47.170	142.251.42.68	ICMP	106	Echo (ping) request id=0x0001, seq=381/32001, ttl=3 (no response found!)
9	11.537241	10.7.47.170	142.251.42.68	ICMP	106	Echo (ping) request id=0x0001, seq=382/32257, ttl=3 (no response found!)
10	17.115309	10.7.47.170	142.251.42.68	ICMP	106	Echo (ping) request id=0x0001, seq=383/32513, ttl=4 (no response found!)
11	17.132871	10.7.47.170	142.251.42.68	ICMP	106	Echo (ping) request id=0x0001, seq=384/32769, ttl=4 (no response found!)
12	17.146353	10.7.47.170	142.251.42.68	ICMP	106	Echo (ping) request id=0x0001, seq=385/33025, ttl=4 (no response found!)
13	22.703142	10.7.47.170	142.251.42.68	ICMP	106	Echo (ping) request id=0x0001, seq=386/33281, ttl=5 (no response found!)
14	22.720665	10.7.47.170	142.251.42.68	ICMP	106	Echo (ping) request id=0x0001, seq=387/33537, ttl=5 (no response found!)
15	22.736131	10.7.47.170	142.251.42.68	ICMP	106	Echo (ping) request id=0x0001, seq=388/33793, ttl=5 (no response found!)
16	28.341225	10.7.47.170	142.251.42.68	ICMP	106	Echo (ping) request id=0x0001, seq=389/34049, ttl=6 (no response found!)
17	28.355905	10.7.47.170	142.251.42.68	ICMP	106	Echo (ping) request id=0x0001, seq=390/34305, ttl=6 (no response found!)
18	28.368693	10.7.47.170	142.251.42.68	ICMP	106	Echo (ping) request id=0x0001, seq=391/34561, ttl=6 (no response found!)
19	33.955412	10.7.47.170	142.251.42.68	ICMP	106	Echo (ping) request id=0x0001, seq=392/34817, ttl=7 (no response found!)
20	33.980171	10.7.47.170	142.251.42.68	ICMP	106	Echo (ping) request id=0x0001, seq=393/35073, ttl=7 (no response found!)
21	34.000505	10.7.47.170	142.251.42.68	ICMP	106	Echo (ping) request id=0x0001, seq=394/35329, ttl=7 (no response found!)
22	39.575900	10.7.47.170	142.251.42.68	ICMP	106	Echo (ping) request id=0x0001, seq=395/35585, ttl=8 (no response found!)
23	39.594140	10.7.47.170	142.251.42.68	ICMP	106	Echo (ping) request id=0x0001, seq=396/35841, ttl=8 (no response found!)
24	39.609107	10.7.47.170	142.251.42.68	ICMP	106	Echo (ping) request id=0x0001, seq=397/36097, ttl=8 (no response found!)
25	45.245717	10.7.47.170	142.251.42.68	ICMP	106	Echo (ping) request id=0x0001, seq=398/36353, ttl=9 (no response found!)
26	45.262161	10.7.47.170	142.251.42.68	ICMP	106	Echo (ping) request id=0x0001, seq=399/36609, ttl=9 (no response found!)
27	45.276113	10.7.47.170	142.251.42.68	ICMP	106	Echo (ping) request id=0x0001, seq=400/36865, ttl=9 (no response found!)
28	50.889181	10.7.47.170	142.251.42.68	ICMP	106	Echo (ping) request id=0x0001, seq=401/37121, ttl=10 (no response found!)
29	50.907885	10.7.47.170	142.251.42.68	ICMP	106	Echo (ping) request id=0x0001, seq=402/37377, ttl=10 (no response found!)
30	50.921001	10.7.47.170	142.251.42.68	ICMP	106	Echo (ping) request id=0x0001, seq=403/37633, ttl=10 (no response found!)
31	56.491986	10.7.47.170	142.251.42.68	ICMP	106	Echo (ping) request id=0x0001, seq=404/37889, ttl=11 (reply in 32)
32	56.507160	142.251.42.68	10.7.47.170	ICMP	106	Echo (ping) reply id=0x0001, seq=404/37889, ttl=115 (request in 31)
33	56.509528	10.7.47.170	142.251.42.68	ICMP	106	Echo (ping) request id=0x0001, seq=405/38145, ttl=11 (reply in 34)
34	56.524402	142.251.42.68	10.7.47.170	ICMP	106	Echo (ping) reply id=0x0001, seq=405/38145, ttl=115 (request in 33)
35	56.530356	10.7.47.170	142.251.42.68	ICMP	106	Echo (ping) request id=0x0001, seq=406/38401, ttl=11 (reply in 36)
36	56.544813	142.251.42.68	10.7.47.170	ICMP	106	Echo (ping) reply id=0x0001, seq=406/38401, ttl=115 (request in 35)

Windows `traceroute` output to www.google.com.

2.5 Observations and Answers

Q1: Protocols used by default

- **Windows tracert:** Uses **ICMP Echo Request** packets with increasing TTL.
- **Linux/macOS traceroute:** Uses **UDP probes to high-numbered ports** by default, unless invoked with options like `-I` (ICMP) or `-T` (TCP).



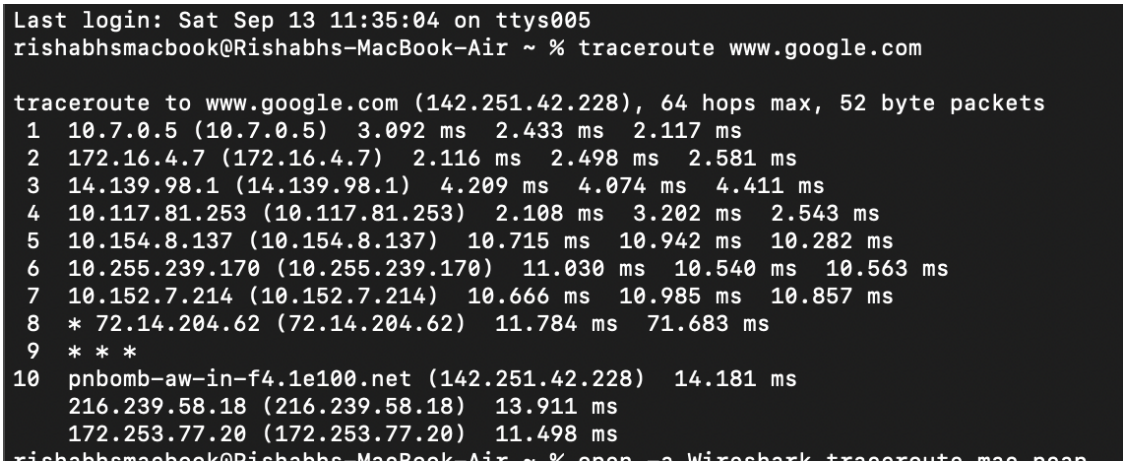
No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	10.240.20.123	142.250.183.100	UDP	66	56904 → 33435 Len=24
2	0.066164	10.240.20.123	142.250.183.100	UDP	66	56904 → 33436 Len=24
3	0.069944	10.240.20.123	142.250.183.100	UDP	66	56904 → 33437 Len=24
4	0.073211	10.240.20.123	142.250.183.100	UDP	66	56904 → 33438 Len=24
5	0.134622	10.240.20.123	142.250.183.100	UDP	66	56904 → 33439 Len=24
6	0.139123	10.240.20.123	142.250.183.100	UDP	66	56904 → 33440 Len=24
7	0.142638	10.240.20.123	142.250.183.100	UDP	66	56904 → 33441 Len=24
8	0.152409	10.240.20.123	142.250.183.100	UDP	66	56904 → 33442 Len=24
9	0.155965	10.240.20.123	142.250.183.100	UDP	66	56904 → 33443 Len=24
10	0.160341	10.240.20.123	142.250.183.100	UDP	66	56904 → 33444 Len=24
11	0.169651	10.240.20.123	142.250.183.100	UDP	66	56904 → 33445 Len=24
12	0.174289	10.240.20.123	142.250.183.100	UDP	66	56904 → 33446 Len=24
13	0.177740	10.240.20.123	142.250.183.100	UDP	66	56904 → 33447 Len=24
14	0.195579	10.240.20.123	142.250.183.100	UDP	66	56904 → 33448 Len=24
15	0.202068	10.240.20.123	142.250.183.100	UDP	66	56904 → 33449 Len=24
16	0.207575	10.240.20.123	142.250.183.100	UDP	66	56904 → 33450 Len=24
17	0.223822	10.240.20.123	142.250.183.100	UDP	66	56904 → 33451 Len=24
18	0.228479	10.240.20.123	142.250.183.100	UDP	66	56904 → 33452 Len=24
19	0.233339	10.240.20.123	142.250.183.100	UDP	66	56904 → 33453 Len=24
20	0.311711	10.240.20.123	142.250.183.100	UDP	66	56904 → 33454 Len=24
21	0.386841	10.240.20.123	142.250.183.100	UDP	66	56904 → 33455 Len=24
22	0.459728	10.240.20.123	142.250.183.100	UDP	66	56904 → 33456 Len=24

macOS `traceroute` output to www.google.com.

Q2: Why might some hops show * * *?

1. **Firewall / ACL blocking traceroute probes:** A firewall or access-list along the path can drop the traceroute probes (UDP to high ports by default, or ICMP if used) so no reply is generated. This is very common on ISP and corporate networks where filtering is applied for security.

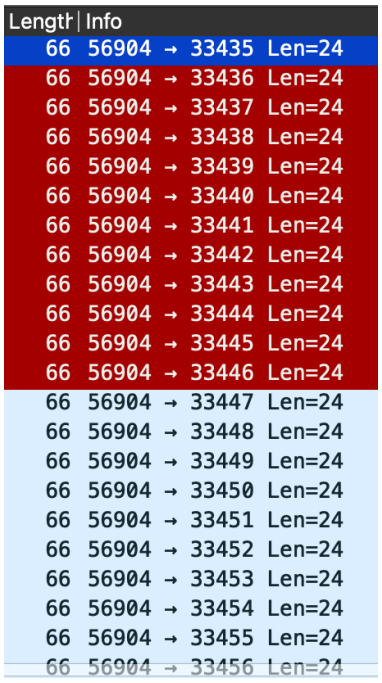
2. **ICMP rate-limiting or deprioritization by routers:** Routers often limit how many ICMP Time-Exceeded/ICMP responses they send per second to protect CPU and control plane resources. When traceroute sends probes faster than that threshold, some replies are dropped and appear as * * *.
3. **Device configured not to respond to TTL-expired probes:** Some devices (or middleboxes) are administratively set not to generate ICMP Time-Exceeded or to not reveal themselves, so they forward traffic but do not respond to traceroute. This is a deliberate configuration to hide topology or because the device operates at L2 and doesn't decrement TTL.
4. **Asymmetric return path or reverse-path filtering:** The probe may reach the hop, but the ICMP reply follows a different return path that is filtered, blocked, or fails, so you never see the response. Asymmetric routing between networks and reverse-path filters in ISPs are common causes of missing hops.



Example of missing hops (* * *) at row 9 in traceroute output.

Q3: Which field changes between successive probes in Linux/macOS traceroute?

In Linux/macOS traceroute, the UDP destination port field changes between successive probes. As shown in the capture, the port increments from 33435 upward, ensuring that the destination host eventually responds with an ICMP "Port Unreachable.



Wireshark capture showing changing destination port values across traceroute probes.

Q4: Difference between final hop and intermediate hops

On Windows, tracert sends ICMP Echo Requests.

- **Intermediate hops:** Intermediate hops respond with ICMP Time Exceeded messages when the TTL expires, but in practice many routers drop or filter these, so the capture may not show them.
- **Final Hop:** Final hop responds differently — instead of Time Exceeded, it sends an ICMP Echo Reply, indicating the destination was reached successfully.

We can see "Echo Reply" final hop response in the following image:

31	56.491986	10.7.47.170	142.251.42.68	ICMP	106 Echo (ping) request id=0x0001, seq=404/37889, ttl=11 (reply in 32)
32	56.507160	142.251.42.68	10.7.47.170	ICMP	106 Echo (ping) reply id=0x0001, seq=404/37889, ttl=115 (request in 31)
33	56.509528	10.7.47.170	142.251.42.68	ICMP	106 Echo (ping) request id=0x0001, seq=405/38145, ttl=11 (reply in 34)
34	56.524402	142.251.42.68	10.7.47.170	ICMP	106 Echo (ping) reply id=0x0001, seq=405/38145, ttl=115 (request in 33)
35	56.530356	10.7.47.170	142.251.42.68	ICMP	106 Echo (ping) request id=0x0001, seq=406/38401, ttl=11 (reply in 36)
36	56.544813	142.251.42.68	10.7.47.170	ICMP	106 Echo (ping) reply id=0x0001, seq=406/38401, ttl=115 (request in 35)

Final ICMP Echo Reply from Google (Windows tracert).

On macOS, traceroute by default sends UDP probes to high-numbered ports.

- **Intermediate hops:** Intermediate hops are expected to send ICMP Time Exceeded, but in practice many routers drop or filter these, so the capture may not show them.
- **Final hops:** Final hop responds with ICMP Destination Unreachable (Port Unreachable), since the chosen UDP port is not open on the destination.

Thus, unlike Windows, the final hop is marked by a Port Unreachable message rather than an Echo Reply.

This behavior is clearly visible in the capture (see highlighted Packet 56 in the screenshot below), where the destination 142.250.183.100 (Google) responds with ICMP Port Unreachable. This marks the successful end of the traceroute.

52	26.602138	10.240.20.123	142.250.183.100	UDP	66 56904 → 33471 Len=24
53	26.766487	10.240.20.123	142.250.183.100	UDP	66 56904 → 33472 Len=24
54	26.916755	10.240.20.123	142.250.183.100	UDP	66 56904 → 33473 Len=24
55	27.058008	10.240.20.123	142.250.183.100	UDP	66 56904 → 33474 Len=24
56	27.132677	142.250.183.100	10.240.20.123	ICMP	70 Destination unreachable (Port unreachable)

Final ICMP Destination Unreachable from Google (macOS traceroute).

Q5: Suppose a firewall blocks UDP traffic but allows ICMP — how would this affect the results of Linux traceroute vs. Windows tracert?

Upon blocking UDP traffic but allowing TCP traffic the following can be observed:

- **Linux/macOS traceroute:** Would fail, as UDP probes are dropped by the firewall. Output shows only * * *.
- **Windows tracert:** Works normally, because ICMP Echo Requests are allowed. Intermediate ICMP Time Exceeded and final Echo Reply will still be received.

2.6 Conclusion

Traceroute behavior differs across operating systems due to their default protocols: Windows uses ICMP, while Linux/macOS use UDP. Missing hops occur due to filtering or rate-limiting of ICMP responses. The final hop is identified by a different ICMP message type (Port Unreachable or Echo Reply). Firewalls selectively blocking UDP or ICMP strongly influence which variant of traceroute succeeds.

2.7 References

[1] Wireshark Documentation
[2] Cisco: Understanding Traceroute Commands