Computer Networks Assignment1

Pavan Deekshith Doddi - 22110190
Kondam Sriman Reddy - 22110124

Task-1: DNS Resolver

Objective

The system is required to parse DNS query packets from a given PCAP file, modify these packets by prepending a custom timestamped header, and send them to a dedicated server. The server, in turn, must resolve these queries not by domain name, but by applying a dynamic, time-based routing algorithm to the custom header to select an IP address from a predefined pool. The final result is a report table that logs each query, its unique header, and the dynamically resolved IP address.

System Architecture

The system consists of two main components: a **Client** and a **Server**, which communicate over UDP. The workflow is as follows:

1. **Input:** The Client program starts by reading a raw network capture (`.pcap`) file.
2. **Client-Side Processing:** It parses the file, isolates each DNS query packet, and generates a unique 8-byte "HHMMSSID" header based on the current time and a sequence number. This header is prepended to the original DNS packet.
3. **Communication:** The newly formed packet is sent to the Server via a UDP socket.
4. **Server-Side Logic:** The Server listens for incoming packets. It extracts the 8-byte custom header and applies a set of predefined, time-based rules to determine a response IP from a load-balancing pool.
5. **Response:** The Server sends the selected IP address back to the Client.
6. **Output:** The Client receives the IP address and logs the complete transaction (Domain Name, Custom Header, Resolved IP) in a formatted table on the console.

Implementation Details

The Client (`client.cpp`)

The client is responsible for reading, modifying, and communicating packets.

- **PCAP Parsing:** The **PcapPlusPlus** library was used to reliably read and parse the input `4.pcap` file. The program iterates through each packet, identifying DNS layers and filtering specifically for query packets (where the QR flag is 0).
- **Custom Header Generation:** A helper function, `createCustomHeader()`, was implemented to generate the "HHMMSSID" header. It uses the `<chrono>` library to get the current system time and formats it along with a sequential packet ID.
- **Communication Protocol:** Standard POSIX sockets were used for UDP communication. The client sends the modified packet to the server at `127.0.0.1:9000` and waits for a response.

PreProcessing

When the client was first run on the raw `4.pcap` file, the output contained a large volume of unexpected DNS queries. This traffic, while valid, was primarily local network "chatter" not relevant to the project's goal of resolving public domain queries.

The identified noise included:

- **mDNS (Multicast DNS):** Queries ending in `.local`, such as `Brother MFC-7860DW._pdl-datastream._tcp.local`, used for discovering devices on the local network.
- **WPAD (Web Proxy Auto-Discovery):** Queries for a hostname named `wpad`, used by clients to find proxy configuration settings.
- **ISATAP:** Queries for a hostname named `isatap`, related to IPv6-to-IPv4 tunneling.

To create a clean dataset for the final report, this traffic was manually filtered out by pre-processing the `4.pcap` file using `tcpdump`. The following command was used to create a new file, `clean_queries.pcap`, that excludes this noise

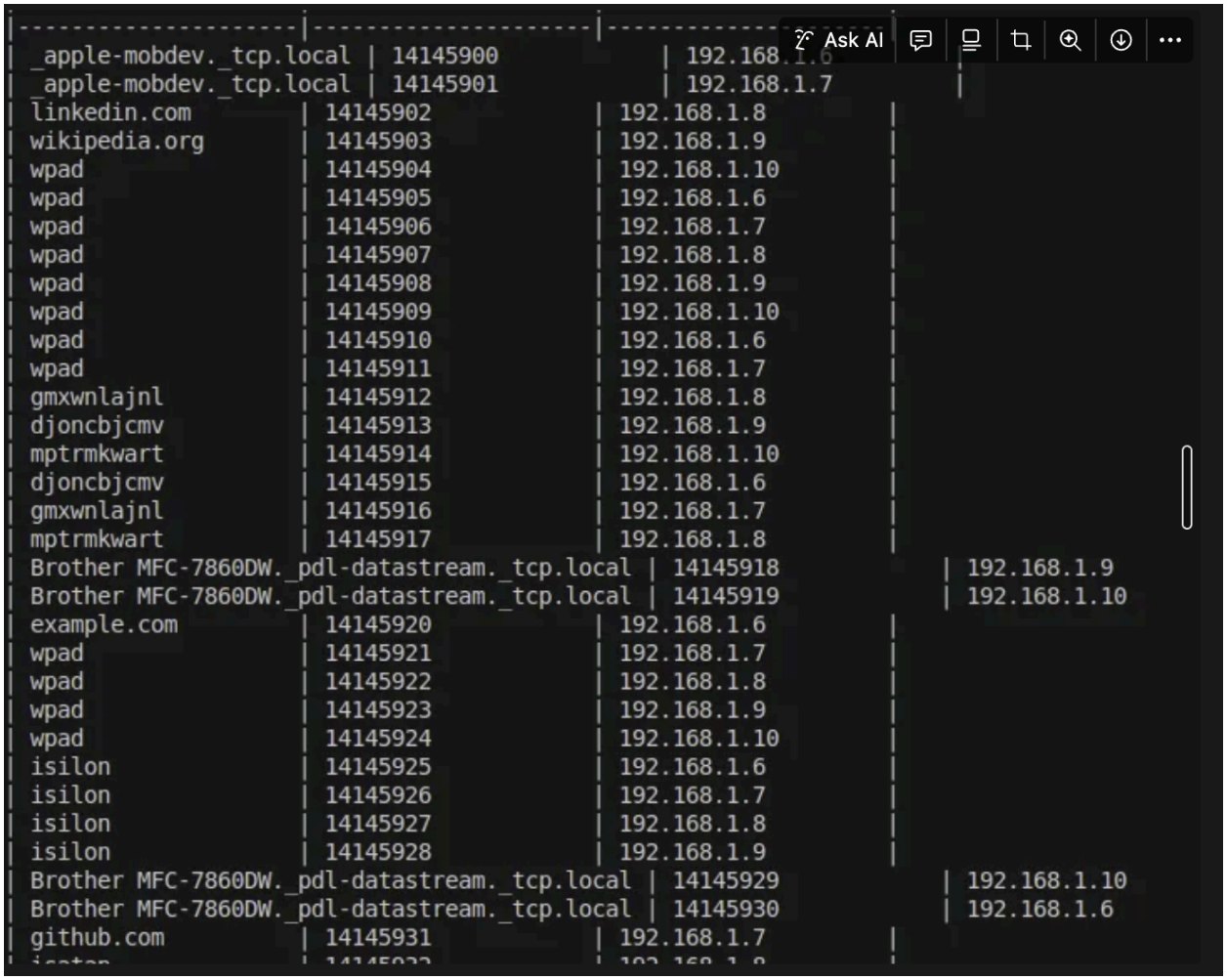

Table before filtering

The Server (`server.cpp`)

The server's core responsibility is to apply the dynamic routing rules.

- **Listening for Connections:** The server binds to port 9000 on all interfaces (`INADDR_ANY`) and enters an infinite loop to listen for client packets using `recvfrom()`.

- **Resolution Algorithm:** Upon receiving a packet, the server extracts the 8-byte header. The logic then proceeds as follows:

    1. The **Hour (HH)** and **ID** are parsed from the header string.
    2. An `if-else if-else` block checks the hour to determine the time of day (morning, afternoon, or night).
    3. Based on the time, a starting index (`ip_pool_start`) for the IP pool is selected.

4. The final IP is chosen using the formula: `final_index = ip_pool_start + (ID % 5)`.

Server Logs: The server was programmed to log every packet it receives to the console, showing the custom header and the IP it resolved. This was crucial for debugging and verifying its behaviour.

```
(base) xiaofeng@angel:~/projects/CN$ ./server
✅ Server is listening on port 9000 with dynamic routing rules...
Received Header: 14174300, Resolved IP: 192.168.1.6
Received Header: 14174301, Resolved IP: 192.168.1.7
Received Header: 14174302, Resolved IP: 192.168.1.8
Received Header: 14174303, Resolved IP: 192.168.1.9
Received Header: 14174304, Resolved IP: 192.168.1.10
Received Header: 14174305, Resolved IP: 192.168.1.6
Received Header: 14174306, Resolved IP: 192.168.1.7
Received Header: 14174307, Resolved IP: 192.168.1.8
Received Header: 14174308, Resolved IP: 192.168.1.9
Received Header: 14174309, Resolved IP: 192.168.1.10
Received Header: 14174310, Resolved IP: 192.168.1.6
Received Header: 14174311, Resolved IP: 192.168.1.7
Received Header: 14174412, Resolved IP: 192.168.1.8
```

Results

The client program was executed on the filtered `clean_queries.pcap` file. The server successfully applied its time-based routing algorithm to resolve the DNS queries. The final report generated by the client is displayed below.

```
(base) xiaofeng@angel:~/projects/CN$ ./client 4.pcap
| Queried Domain Name | Custom Header Value | Resolved IP Address |
|---------------------|---------------------|---------------------|
| linkedin.com        | 14174300            | 192.168.1.6         |
| wikipedia.org       | 14174301            | 192.168.1.7         |
| gmxwnlajnl          | 14174302            | 192.168.1.8         |
| djoncbjcmv          | 14174303            | 192.168.1.9         |
| mptrmkwart          | 14174304            | 192.168.1.10        |
| djoncbjcmv          | 14174305            | 192.168.1.6         |
| gmxwnlajnl          | 14174306            | 192.168.1.7         |
| mptrmkwart          | 14174307            | 192.168.1.8         |
| example.com         | 14174308            | 192.168.1.9         |
| github.com          | 14174309            | 192.168.1.10        |
| reddit.com          | 14174310            | 192.168.1.6         |
| google.com          | 14174311            | 192.168.1.7         |
| bing.com            | 14174412            | 192.168.1.8         |
```

The final results table includes several queries for seemingly random hostnames, such as `gmxwnlajnl` and `djoncbjcmv`. These did not appear in the initial `tcpdump` inspection.

Unlike the mDNS or WPAD traffic, these queries are standard unicast DNS queries sent to an internet-based DNS server. They are typically generated automatically by modern web browsers for security checks (e.g., detecting DNS hijacking) or performance (e.g., DNS prefetching). Therefore, these queries were correctly preserved by our filtering process and processed by the client, as they represent legitimate DNS traffic found within the capture file.

Task-2: Traceroute Protocol Behavior

Question 1:

Analysis of macOS `traceroute` using Wireshark

The `traceroute` utility on Linux and macOS systems operates using a combination of UDP for its outgoing probes and ICMP for the incoming replies. This process is designed to elicit a specific error message from each router along the path.

1. Outgoing Probes: UDP Packets

The command initiates the trace by sending a sequence of UDP packets to the destination IP address. A key detail of this process is that for each **hop** (each router in the path), it sends **three separate probe packets**. This is done to provide a more stable and accurate measurement of the round-trip time (RTT) to that hop, as network conditions can fluctuate. The command then displays the timings for each of the three probes and often an average.

The first set of three packets is sent with an IP **Time to Live (TTL)** value of 1, the second set with a TTL of 2, and so on. This ensures that each successive set of probes travels one hop further down the path.

As shown in the network capture figure 2.1, when filtering for "udp," we can clearly see these outgoing probe packets originating from the source machine and destined for the target server.

2. Incoming Replies: ICMP Packets

When a router along the path receives one of these UDP packets and decrements its TTL value to 0, it discards the packet. It then sends an **ICMP (Internet Control Message Protocol)** packet back to the source machine. Specifically, it sends a **Type 11: "Time-to-live exceeded"** message.

This ICMP reply is the signal `traceroute` uses to identify the router at that specific hop. The capture below, filtered for "UDP" shows these replies coming from the intermediate routers.

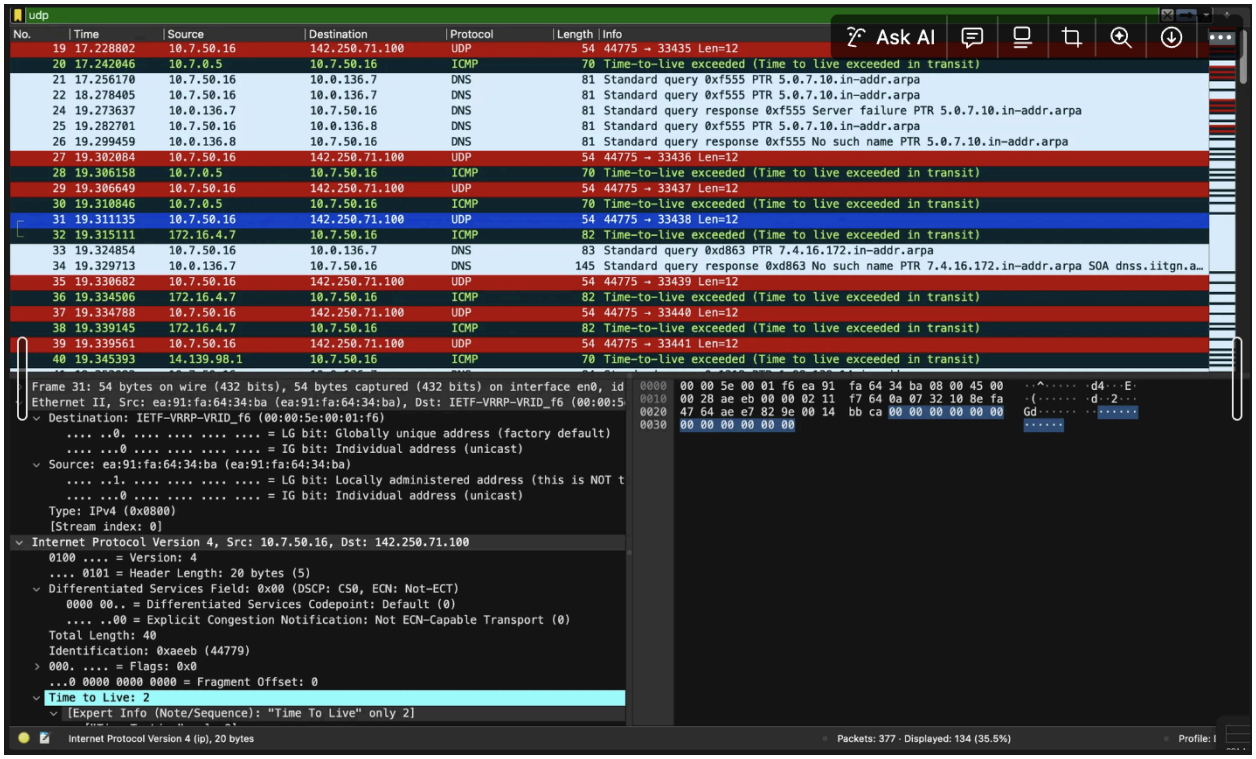

Figure 2.1.1 Macos using wireshark

Analysis of Linux `traceroute` using TCPDUMP

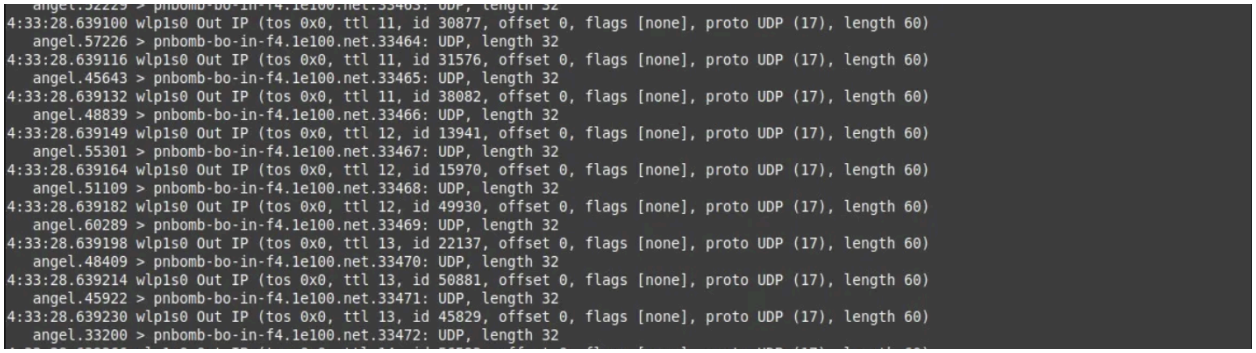

Figure 2.1.2 Linux using tcpdump

Comparison with Windows `tracert`

In contrast, the Windows `tracert` utility does not use UDP packets. Instead, it sends **ICMP Echo Request** packets for its probes—the same type of packet used by the `ping` command.

The intermediate routers still respond with **ICMP "Time-to-live exceeded"** messages. When the final destination is reached, it responds with an **ICMP Echo Reply**.

Conclusion:

Operating System	Command	Outgoing Probe Protocol	Incoming Reply Protocol
Linux/macOS	`traceroute`	**UDP**	**ICMP**("Time-to-live exceeded")
Windows	`tracert`	**ICMP**("EchoRequest")	**ICMP**("Time-to-live exceeded")

Question 2:

The output of a `traceroute` command sometimes displays asterisks (`* * *`) for a particular hop, indicating that no reply was received from the router at that position within the given timeout period. The provided `traceroute` capture to `www.google.com` demonstrates this phenomenon clearly at **Hop 9.**

```
(base) pavandeekshith@Pavans-MacBook-Air-3613 ~ % traceroute www.google.com
traceroute to www.google.com (142.250.71.100), 64 hops max, 40 byte packets
 1  10.7.0.5 (10.7.0.5)  11.149 ms  4.774 ms  4.186 ms
 2  172.16.4.7 (172.16.4.7)  4.642 ms  4.622 ms  4.302 ms
 3  14.139.98.1 (14.139.98.1)  8.281 ms  6.317 ms  6.321 ms
 4  10.117.81.253 (10.117.81.253)  13.394 ms  4.211 ms  4.334 ms
 5  10.154.8.137 (10.154.8.137)  13.161 ms  12.781 ms  12.694 ms
 6  10.255.239.170 (10.255.239.170)  13.271 ms  12.390 ms  11.847 ms
 7  10.152.7.214 (10.152.7.214)  12.625 ms  12.638 ms  12.896 ms
 8  72.14.204.62 (72.14.204.62)  13.739 ms * *
 9  * * *
10  192.178.86.200 (192.178.86.200)  17.706 ms
    142.250.227.72 (142.250.227.72)  17.972 ms
    142.250.238.80 (142.250.238.80)  13.476 ms
11  192.178.110.106 (192.178.110.106)  14.848 ms
    192.178.110.198 (192.178.110.198)  17.284 ms
    192.178.110.208 (192.178.110.208)  15.007 ms
12  pnbomb-ad-in-f4.1e100.net (142.250.71.100)  24.895 ms  23.798 ms  24.051 ms
(base) pavandeekshith@Pavans-MacBook-Air-3613 ~ %
```

Figure 2.2.1 MacOS using Wireshark

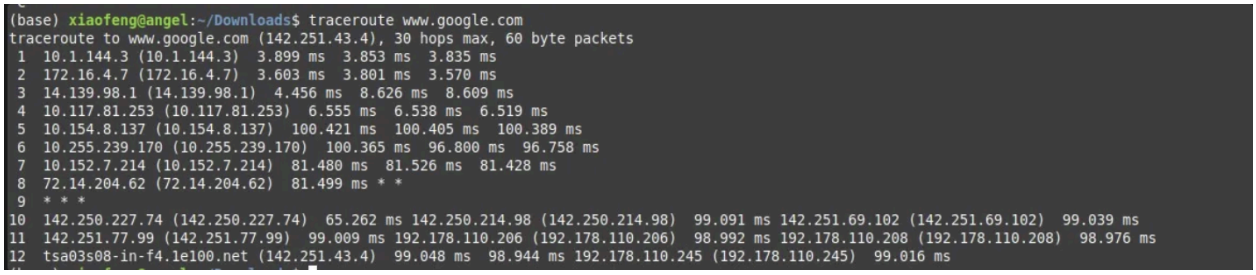

Figure 2.2.2 Linux Using TCPDUMP

This lack of response does not necessarily mean the router is down; rather, it is often a deliberate configuration choice or a sign of network congestion. There are two primary reasons why a router might not reply:

1. **Firewall or Access Control List (ACL) Filtering:** This is the most common reason. For security purposes, network administrators often configure routers or firewalls to block outgoing **ICMP "Time-to-live exceeded"** messages. By preventing these replies, they can obscure their internal network topology from external scanning tools like `traceroute`. The router receives and forwards the probe packet (if the TTL were higher), but it is explicitly forbidden from sending a reply back to the source.
2. **ICMP Rate Limiting:** A router's main function is to forward traffic, not to generate diagnostic replies. To protect its CPU from being overwhelmed by too many diagnostic requests (like those from `traceroute` or `ping`), a router may be configured to limit the rate at which it sends ICMP messages. If the router is busy or receives too many probes in a short period, it will de-prioritize and drop the request to send a reply, resulting in a timeout on the user's end.

Question 3:

In a Linux or macOS `traceroute`, the critical field within the IP header that changes between successive **hops** is the **Time to Live (TTL)**.

This field is intentionally manipulated by the `traceroute` utility. It begins by sending a set of probes with a TTL value of 1 to discover the first router. After receiving a reply, it sends a new set of probes with an incremented TTL value of 2 to discover the second router, and this process continues for each subsequent hop.

The first packet capture below shows an initial probe packet sent by the `traceroute` command. In the detailed view of the IP header, the **Time to Live** field is set to **1**. This ensures the packet will only travel one hop before being stopped by the first router.

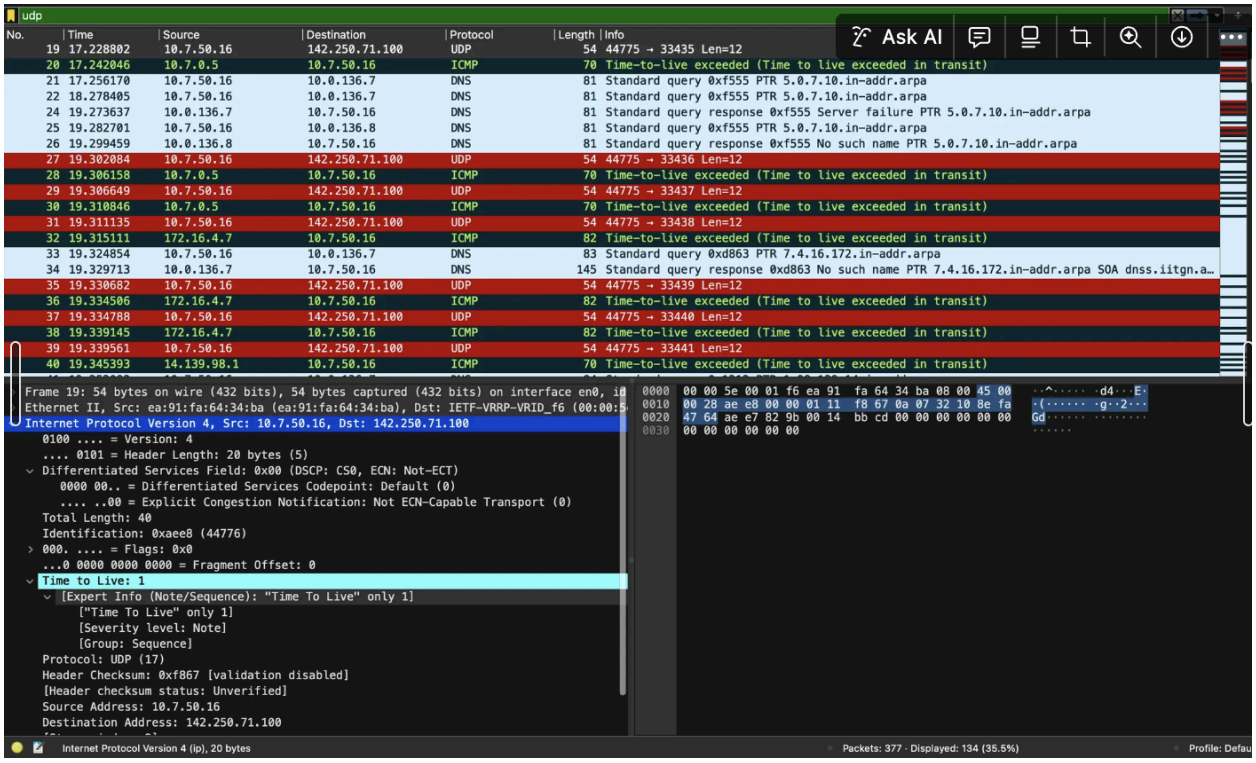

Figure 2.3.1 Macos using Wireshark

The second capture shows a subsequent probe packet destined for the next hop in the sequence. As highlighted, the **Time to Live** field has now been incremented to **2**. This allows the packet to pass through the first router and be stopped by the second, thereby revealing the next step in the path.

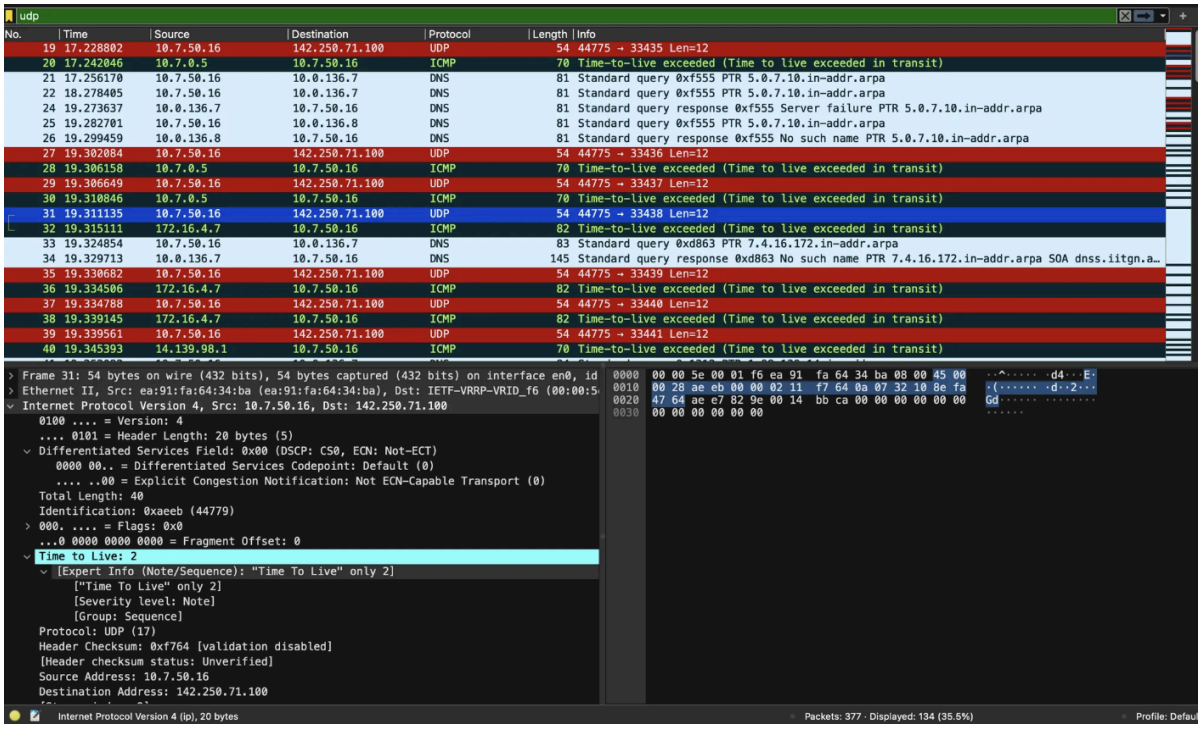

Figure 2.3.2 Macos using Wireshark

This systematic increment of the TTL value is the fundamental mechanism that allows `traceroute` to map the entire route to a destination, one hop at a time.

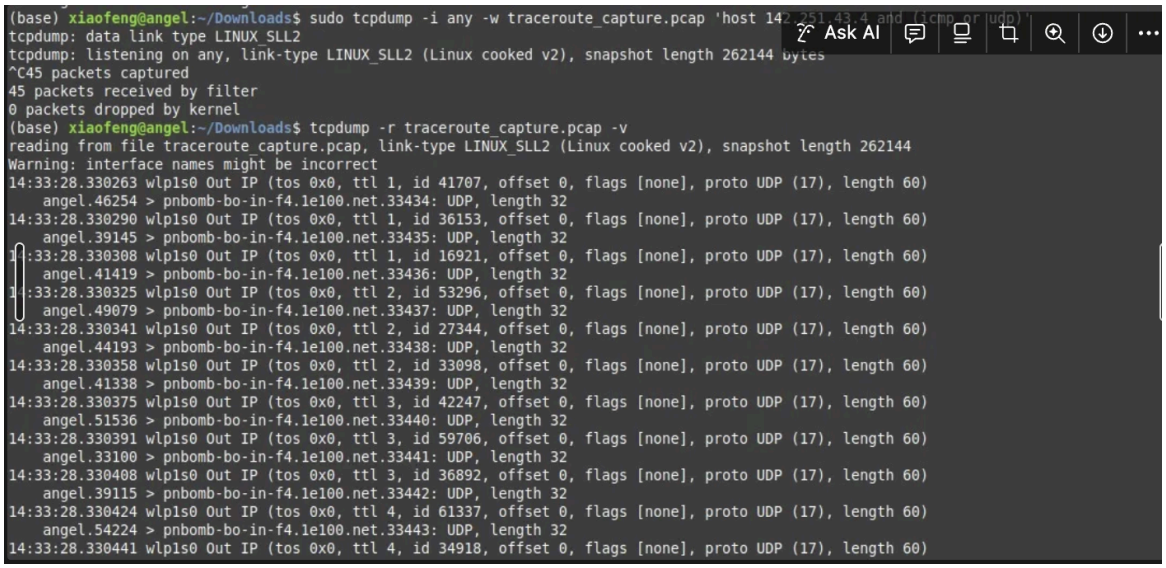

Figure 2.3.3 Linux using tcpdump

As we can see from the image the first 3 UDP calls show ttl = 1, and the 4th call(hop2) shows ttl = 2 showcasing, IP header that changes between successive hops is the Time to Live (TTL)

Question 4:

The response from the final hop in a `traceroute` is fundamentally different from the responses generated by intermediate hops. The difference lies in the specific **type of ICMP message** sent back to the source. Intermediate hops signal their presence with a "Time-to-live exceeded" message, whereas the final destination signals the end of the trace with a "Destination unreachable" message.

Intermediate Hop Response: "Time-to-live exceeded"

For every intermediate hop along the path, the `traceroute` probe packet is sent with a TTL value that is too low to travel any further. The router at that hop decrements the TTL to zero, discards the packet, and sends back an **ICMP Type 11: "Time-to-live exceeded"** message. This message effectively tells the source machine, "I am router X on the path to your destination."

The screenshot below captures this exact behavior. The selected packet is an ICMP reply from an intermediate router (`72.14.204.62`), and the packet list confirms its purpose is to signal that the time to live has been exceeded.

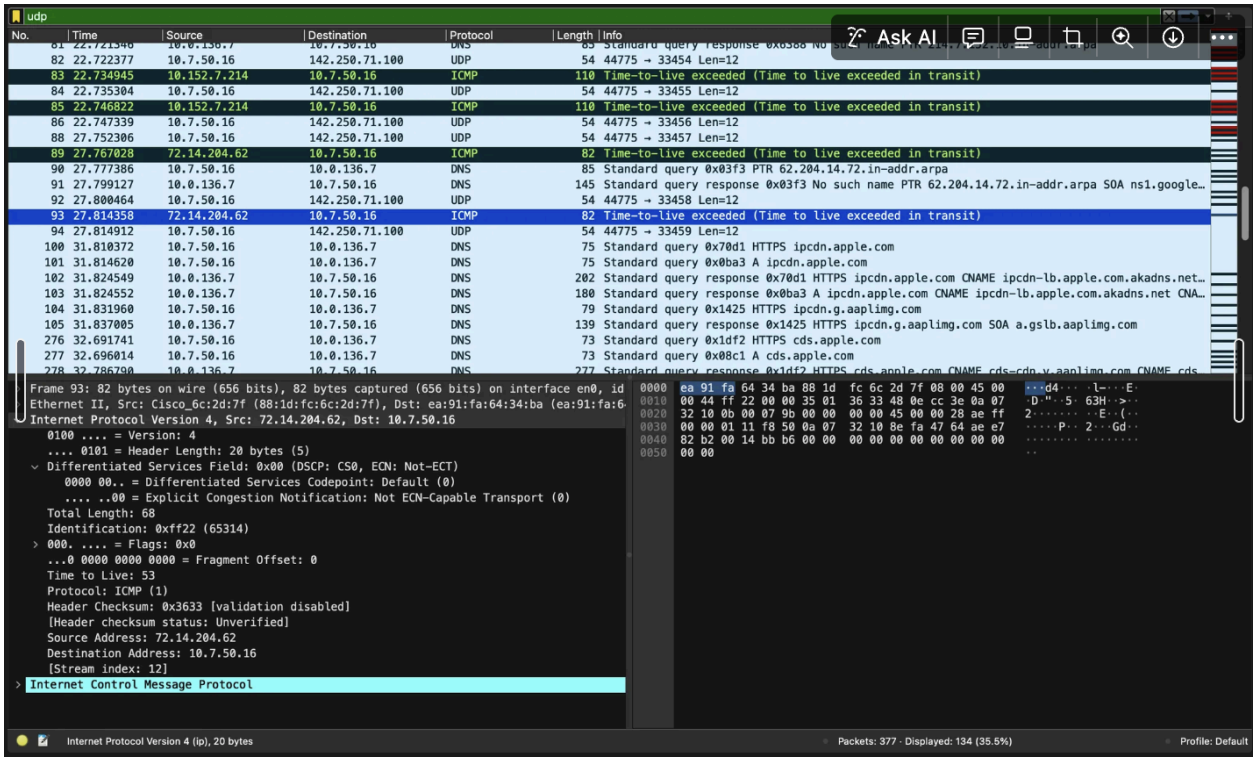

Figure 2.4.1 Macos using Wireshark

Final Hop Response: "Destination unreachable (Port unreachable)"

When the TTL of the probe packet is finally high enough to reach the destination server, a different mechanism occurs. The server receives the **UDP** probe, but it is addressed to a random, high-numbered port on which no service is actively listening.

The server's operating system, seeing an incoming packet for a closed port, responds with an **ICMP Type 3, Code 3: "Destination unreachable (Port unreachable)"** message. This reply serves as the definitive signal that the trace has successfully reached the intended destination.

The second screenshot clearly illustrates this final response. The packet is an ICMP message originating from the final destination (`142.250.71.100`), and its information explicitly states **"Destination unreachable (Port unreachable)"**.

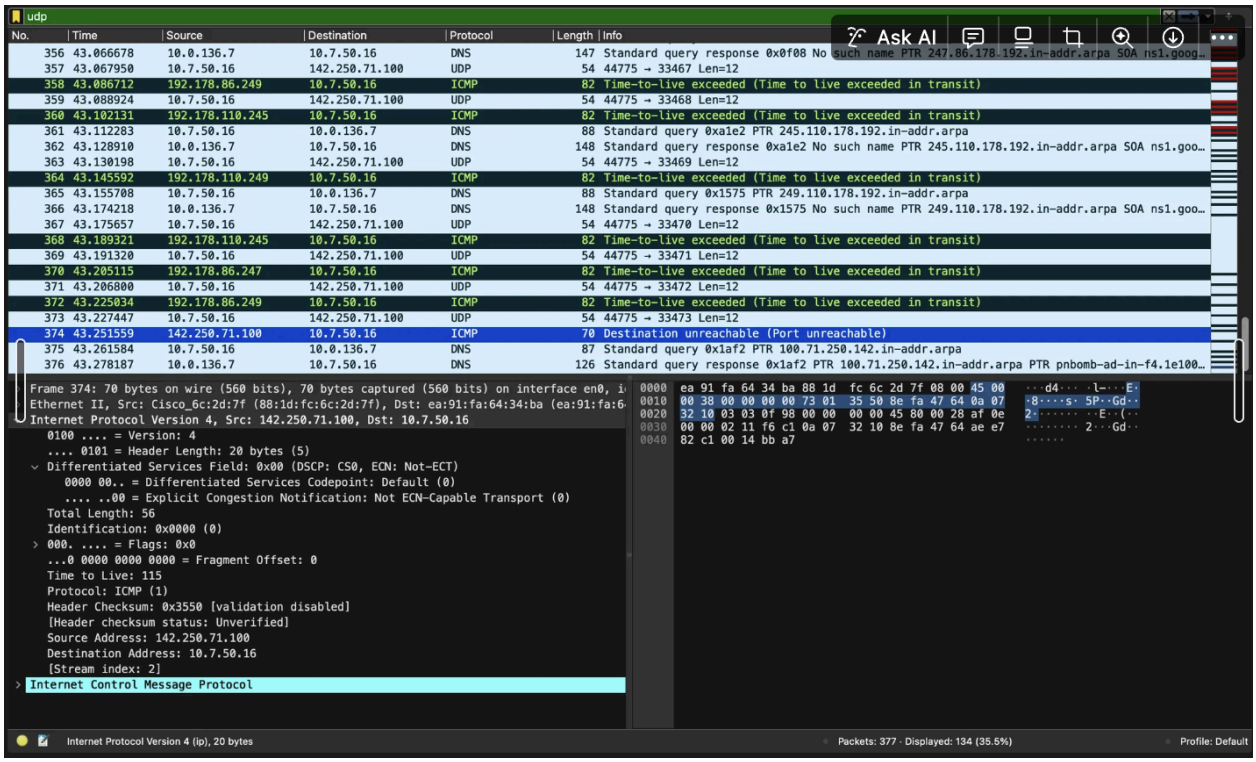

Figure 2.4.2 Macos using Wireshark

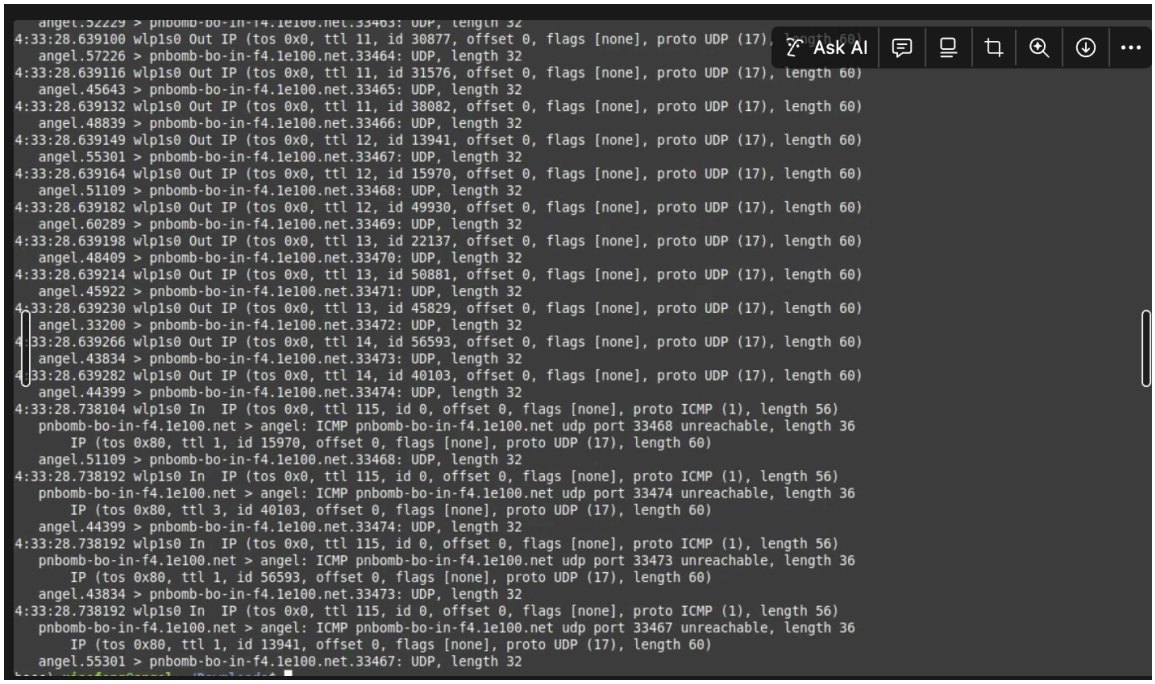

Figure 2.4.3 Linux using tcpdump

In the Linux method, the ICMP message below shows the final response and in the above where ttl is 11,12 they are the intermediate hop responses.

Question 5:

The firewall rule—blocking **UDP** while allowing **ICMP**—would cause the Linux/macOS `traceroute` to fail completely, while the Windows `tracert` would work without any issues.

Effect on Linux/macOS `traceroute`

As established in the previous analysis, the `traceroute` command on Linux and macOS sends **UDP** packets as its outgoing probes.

1. When `traceroute` sends its first UDP probe with TTL=1, the packet will be inspected by the firewall.
2. The firewall, seeing that the packet is UDP, will apply its rule and **BLOCK** the traffic.
3. The packet will never reach the first router, and therefore no ICMP "Time-to-live exceeded" reply will ever be generated.
4. The `traceroute` command will time out waiting for a reply and print a line of asterisks (`* *`).

This process would repeat for every hop, causing the **Linux `traceroute` to fail completely**.

Effect on Windows `tracert`

The `tracert` command on Windows, by contrast, sends **ICMP** Echo Request packets as its outgoing probes.

1. When `tracert` sends its first ICMP probe with TTL=1, the packet will be inspected by the firewall.
2. The firewall, seeing that the packet is ICMP, will apply its rule and **ALLOW** the traffic to pass through.
3. The packet will reach the first router, which will then generate an ICMP "Time-to-live exceeded" reply.
4. This incoming ICMP reply will also be **ALLOWED** back through the firewall.

This process would repeat for every hop, allowing the **Windows `tracert` to work perfectly fine**, as both its outgoing and incoming packets are ICMP.

Conclusion:

Utility	**Probe Protocol**	**Firewall Action**	**Result**
Linux `traceroute`	UDP	**Blocked**	**Fails Completely** (`* * *`)
Windows `tracert`	ICMP	**Allowed**	**Works Perfectly**