Assignment-1 Report

Astitva Aryan (22110048), Dhruv Sharma (22110074)

September 15, 2025

Source Code Repository: `github.com/astitvaaryan/CS-331...`

Task 1

The purpose of this task was to implement a custom DNS resolution system consisting of a client and a server. The client was designed to parse DNS query packets from a given PCAP file [(041 + 074) % 10 = 5] (‘5.pcap‘). For each query, the client generated a custom 8-byte header in the format `HHMMSSID` (Hour, Minute, Second, Sequential ID) and prepended it to the original DNS packet.

The modified packet was then sent to a custom server. The server, instead of performing a live DNS lookup, implemented a set of predefined, time-based rules. It used the hour and ID from the custom header to deterministically assign an IP address from a static pool of 15 addresses. This process simulates a time-based load balancing system.

Methodology

The implementation consists of two Python scripts: a client and a server. The client is responsible for packet parsing and forwarding, while the server handles the custom logic for IP address assignment.

Client Code (client.py)

```
import socket
from datetime import datetime
from scapy.all import rdpcap, DNS, DNSQR, raw

# --- Configuration ---
PCAP_FILE = '5.pcap'
SERVER_IP = '127.0.0.1'
SERVER_PORT = 9999
REPORT_FILE = 'report.txt'

def process_dns_queries():
    """
    Reads a pcap file, filters DNS queries, sends them to the custom server,
    receives the resolved IP, and generates a final report.
    """
    print(f"[*] Reading packets from '{PCAP_FILE}'...")
    try:
        packets = rdpcap(PCAP_FILE)
    except FileNotFoundError:
        print(f"[ERROR] The file '{PCAP_FILE}' was not found. Please place it
    in the same directory.")
```

```
        return

    # Filter for packets that are standard DNS queries (qr=0 means query)
    dns_queries = [pkt for pkt in packets if pkt.haslayer(DNS) and pkt[DNS].qr == 0]

    if not dns_queries:
        print("[*] No DNS queries found in the pcap file.")
        return

    print(f"[*] Found {len(dns_queries)} DNS queries. Processing...")

    report_data = []
    query_id_counter = 0

    for query_packet in dns_queries:
        try:
            # 1. Create the custom 8-byte header (HHMMSSID).
            now = datetime.now()
            hhmmss = now.strftime("%H%M%S")
            # Format ID with a leading zero if it's a single digit (e.g., 00, 01, ..., 09, 10)
            query_id_str = f"{query_id_counter:02d}"
            custom_header = f"{hhmmss}{query_id_str}"

            # 2. Construct the payload: custom header + original DNS packet bytes.
            header_bytes = custom_header.encode('utf-8')
            packet_bytes = raw(query_packet)
            payload = header_bytes + packet_bytes

            # 3. Connect to the server and send the payload.
            with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
                s.connect((SERVER_IP, SERVER_PORT))
                s.sendall(payload)

                # 4. Receive the server's response (the assigned IP address).
                response = s.recv(1024).decode('utf-8')
                assigned_ip = response

            # 5. Log data for the report.
            queried_domain = query_packet[DNSQR].qname.decode('utf-8').rstrip('.')
            report_data.append((custom_header, queried_domain, assigned_ip))
            print(f"  - Sent Query for '{queried_domain}' with Header '{custom_header}' -> Got IP: {assigned_ip}")

            query_id_counter += 1

        except Exception as e:
            print(f"[ERROR] Failed to process a query: {e}")

    # 6. Generate the final report file.
    generate_report(report_data)

def generate_report(data):
    """Writes the final report to a text file and prints it."""
    header = f"{'Custom Header (HHMMSSID)':<30} | {'Domain Name':<40} | {'Assigned IP Address'}\n"
    separator = "-" * 100 + "\n"

    report_content = header + separator

```

```
    for entry in data:
        report_content += f"{entry[0]:<30} | {entry[1]:<40} | {entry[2]}\n"

    try:
        with open(REPORT_FILE, 'w') as f:
            f.write(report_content)
        print(f"\n[*] Report successfully generated: '{REPORT_FILE}'")
    except IOError as e:
        print(f"\n[ERROR] Could not write report to file: {e}")

    print("\n--- Final Report ---")
    print(report_content)


if __name__ == "__main__":
    process_dns_queries()
```

Listing 1: The client script reads DNS queries from a PCAP file, adds a custom header, and sends them to the server for resolution.

Server Code (server.py)

```
import socket
import threading

# Define the static IP pool as specified in the rules document.
IP_POOL = [
    "192.168.1.1", "192.168.1.2", "192.168.1.3", "192.168.1.4", "192.168.1.5",
    "192.168.1.6", "192.168.1.7", "192.168.1.8", "192.168.1.9", "192.168.1.10",
    "192.168.1.11", "192.168.1.12", "192.168.1.13", "192.168.1.14", "
    192.168.1.15"
]

def handle_client(conn, addr):
    """
    Handles an individual client connection, applies the DNS resolution rules,
    and sends back the assigned IP address.
    """
    print(f"[NEW CONNECTION] {addr} connected.")
    try:
        # Receive the payload from the client
        data = conn.recv(2048)
        if not data:
            return

        # 1. Extract the 8-byte custom header from the payload.
        custom_header = data[:8].decode('utf-8')

        # 2. Parse the header to get the hour (HH) and ID.
        hour = int(custom_header[0:2])
        query_id = int(custom_header[6:8])

        # 3. Apply the time-based routing rules to find the starting index.
        ip_pool_start = 0
        if 4 <= hour <= 11:  # Morning: 04:00-11:59
            ip_pool_start = 0
            time_of_day = "Morning"
        elif 12 <= hour <= 19: # Afternoon: 12:00-19:59
            ip_pool_start = 5
            time_of_day = "Afternoon"
        else: # Night: 20:00-03:59
            ip_pool_start = 10
```

```
            time_of_day = "Night"

        # 4. Calculate the offset using the ID modulo 5.
        offset = query_id % 5

        # 5. Calculate the final index and select the IP address.
        final_index = ip_pool_start + offset
        assigned_ip = IP_POOL[final_index]

        print(f"  [PROCESSING] Header: {custom_header} | Hour: {hour} ({time_of_day}) | ID: {query_id}")
        print(f"  [LOGIC] Pool Start: {ip_pool_start} | Offset (ID % 5): {offset} | Final Index: {final_index}")
        print(f"  [RESULT] Assigned IP: {assigned_ip}")

        # 6. Send the assigned IP back to the client.
        conn.sendall(assigned_ip.encode('utf-8'))

    except (ValueError, IndexError) as e:
        print(f"[ERROR] Invalid data received from {addr}: {e}")
        error_message = "Error: Invalid header or data format"
        conn.sendall(error_message.encode('utf-8'))
    finally:
        conn.close()
        print(f"[CONNECTION CLOSED] {addr}")

def start_server(host='127.0.0.1', port=9999):
    """
    Starts the DNS resolver server, listening for incoming connections.
    """
    server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server_socket.bind((host, port))
    server_socket.listen()
    print(f"[LISTENING] Server is listening on {host}:{port}")

    while True:
        conn, addr = server_socket.accept()
        # Create a new thread to handle each client connection so the server can
        # handle multiple requests concurrently without blocking.
        thread = threading.Thread(target=handle_client, args=(conn, addr))
        thread.start()

if __name__ == "__main__":
    start_server()
```

Listing 2: The server script listens for client connections, parses the custom header, and assigns an IP based on predefined time-based rules.

Output

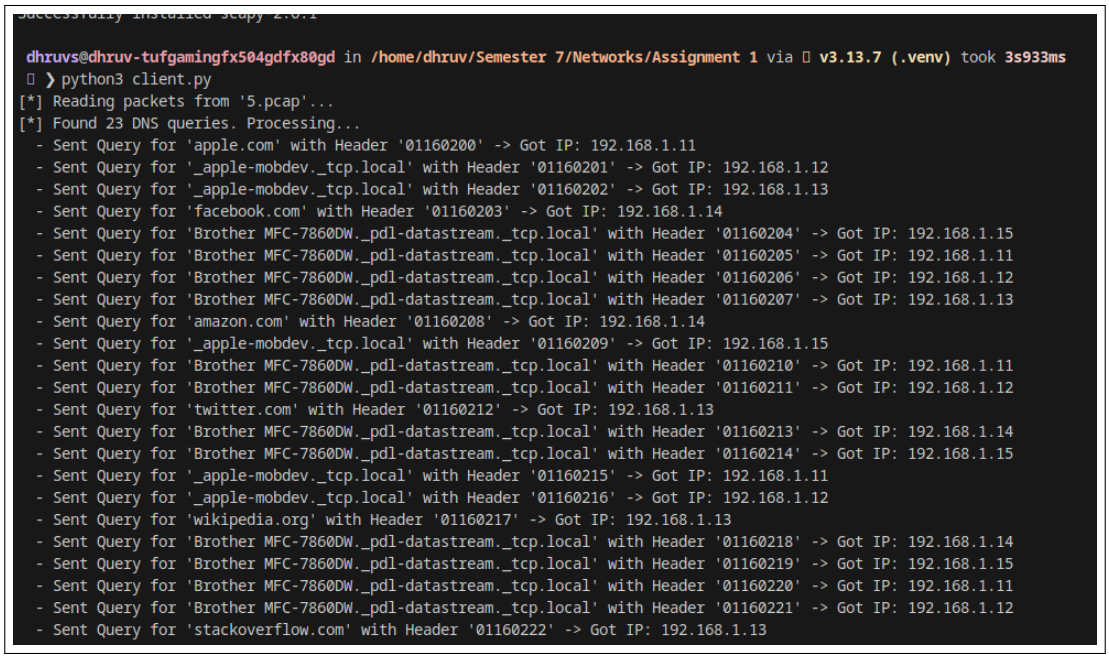

Figure 1: Client Output showing the script processing DNS queries and receiving IPs.

Results

The following table presents the output generated by the client after processing all DNS queries from the PCAP file and receiving the assigned IP addresses from the server.

Custom Header (HHMMSSID)	Domain Name	Assigned IP Add
02110100	apple.com	192.168.1.11
02110101	_apple-mobdev._tcp.local	192.168.1.12
02110102	_apple-mobdev._tcp.local	192.168.1.13
02110103	facebook.com	192.168.1.14
02110104	Brother MFC-7860DW._pdl-datastream._tcp.local	192.168.1.15
02110105	Brother MFC-7860DW._pdl-datastream._tcp.local	192.168.1.11
02110106	Brother MFC-7860DW._pdl-datastream._tcp.local	192.168.1.12
02110107	Brother MFC-7860DW._pdl-datastream._tcp.local	192.168.1.13
02110108	amazon.com	192.168.1.14
02110109	_apple-mobdev._tcp.local	192.168.1.15
02110110	Brother MFC-7860DW._pdl-datastream._tcp.local	192.168.1.11
02110111	Brother MFC-7860DW._pdl-datastream._tcp.local	192.168.1.12
02110112	twitter.com	192.168.1.13

Continued on next p

Table 1 – continued from previous page

Custom Header (HHMMSSID)	Domain Name	Assigned IP Add
02110113	Brother MFC-7860DW._pdl-datastream._tcp.local	192.168.1.14
02110114	Brother MFC-7860DW._pdl-datastream._tcp.local	192.168.1.15
02110115	_apple-mobdev._tcp.local	192.168.1.11
02110116	_apple-mobdev._tcp.local	192.168.1.12
02110117	wikipedia.org	192.168.1.13
02110118	Brother MFC-7860DW._pdl-datastream._tcp.local	192.168.1.14
02110119	Brother MFC-7860DW._pdl-datastream._tcp.local	192.168.1.15
02110120	Brother MFC-7860DW._pdl-datastream._tcp.local	192.168.1.11
02110121	Brother MFC-7860DW._pdl-datastream._tcp.local	192.168.1.12
02110122	stackoverflow.com	192.168.1.13

Task-2: Traceroute Protocol Behavior

The purpose of this task is to understand how the traceroute utility works in different operating systems by capturing and analyzing the network traffic it generates.

Experiment on Windows

The `tracert` command was executed on Windows to trace the route to `www.google.com`. The network traffic was captured simultaneously using Wireshark.

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

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

  1     5 ms     5 ms     7 ms  10.7.0.5
  2     6 ms     5 ms     4 ms  172.16.4.7
  3    13 ms    11 ms    17 ms  14.139.98.1
  4     6 ms     4 ms     5 ms  10.117.81.253
  5    36 ms    15 ms    15 ms  10.154.8.137
  6    15 ms    16 ms    14 ms  10.255.239.170
  7    13 ms    12 ms    11 ms  10.152.7.214
  8    12 ms    13 ms    12 ms  72.14.204.62
  9    13 ms    15 ms    14 ms  72.14.239.103
 10    15 ms    14 ms    14 ms  142.251.69.105
 11    20 ms    16 ms    18 ms  bom12s21-in-f4.1e100.net [142.251.42.68]

Trace complete.
```

Figure 2: Output of `tracert www.google.com` on Windows.

Experiment on Linux

The purpose of this experiment was to analyze the behavior of the ‘traceroute‘ utility on a Linux-based operating system (Arch Linux). The process involved running a traceroute to a public destination (‘google.com‘) while simultaneously capturing the generated network traffic using ‘tcpdump‘. The captured data is used to answer fundamental questions about the tool’s underlying protocol mechanics.

Methodology: Execution and Capture

The analysis was performed by running ‘traceroute google.com‘ in one terminal and ‘sudo tcpdump -i wlo1 -n -vv ’host google.com or icmp’‘ in a second terminal. This allowed for a live capture of both the outgoing probes sent by the client and the incoming ICMP responses from routers along the path.

Traceroute Output

The following is the terminal output from the ‘traceroute‘ command. It successfully maps the path to the destination, showing a failure at hop 9.

```
traceroute to google.com (142.251.222.78), 30 hops max, 60 byte packets
1  10.1.144.3 (10.1.144.3)  2.815 ms  2.792 ms  2.781 ms
2  172.16.4.7 (172.16.4.7)  2.062 ms  2.756 ms  2.744 ms
3  14.139.98.1 (14.139.98.1)  3.861 ms  5.453 ms  5.435 ms
4  10.117.81.253 (10.117.81.253)  2.667 ms  2.620 ms  2.596 ms
5  10.154.8.137 (10.154.8.137)  11.950 ms  13.502 ms  12.397 ms
6  10.255.239.170 (10.255.239.170)  15.985 ms  14.107 ms  13.614 ms
7  10.152.7.214 (10.152.7.214)  9.616 ms  9.607 ms  9.597 ms
8  72.14.204.62 (72.14.204.62)  14.436 ms  14.422 ms *
9  * * *
10  192.178.86.200 (192.178.86.200)  16.134 ms 142.251.77.100 (142.251.77.100)
     38.084 ms  38.075 ms
11  192.178.110.208 (192.178.110.208)  10.357 ms 192.178.110.104
    (192.178.110.104)  11.596 ms 192.178.110.110 (192.178.110.110)  38.046 ms
12  142.250.208.227 (142.250.208.227)  14.906 ms pnbomb-bp-in-f14.1e100.net
    (142.251.222.78)  10.806 ms 142.250.209.71 (142.250.209.71)  12.880 ms
```

Listing 3: Terminal output of the traceroute command to google.com.

Tcpdump Packet Capture

The following is an abridged ‘tcpdump‘ capture that highlights the key packets involved in the traceroute process.

```
tcpdump: listening on wlo1, link-type EN10MB (Ethernet), snapshot length 262144
     bytes

% --- Probe for Hop 1 and its reply ---
01:35:40.916080 IP (tos 0x0, ttl 255, id 61206, proto ICMP (1), length 56)
    10.1.144.3 > 10.1.144.19: ICMP time exceeded in-transit, length 36
        IP (tos 0x0, ttl 1, id 61206, proto UDP (17), length 60)
            10.1.144.19.60238 > 142.251.222.78.33434: UDP, length 32
...
% --- Probe for Hop 2 and its reply ---
01:35:40.929641 IP (tos 0x0, ttl 248, id 63877, proto ICMP (1), length 168)
    10.255.239.170 > 10.1.144.19: ICMP time exceeded in-transit, length 148
        IP (tos 0x0, ttl 2, id 42881, proto UDP (17), length 60)
            10.1.144.19.38871 > 142.251.222.78.33449: [udp sum ok] UDP, length
    32
...
```

```
% --- Reply from the Final Destination ---
01:35:41.053762 IP (tos 0x80, ttl 115, id 0, proto ICMP (1), length 56)
    142.251.222.78 > 10.1.144.19: ICMP 142.251.222.78 udp port 33474
    unreachable, length 36
        IP (tos 0x80, ttl 3, id 48364, proto UDP (17), length 60)
            10.1.144.19.49232 > 142.251.222.78.33474: UDP, length 32

39 packets captured
```

Listing 4: Tcpdump capture showing UDP probes and ICMP replies.

Analysis and Questions

The following questions are answered based on the captured traffic and command-line output.

1. **What protocol does Windows tracert use by default, and what protocol does Linux traceroute use by default?**

   Windows `tracert` uses the **ICMP** (Internet Control Message Protocol) by default. It sends a series of ICMP Echo Request packets with increasing TTL values.

   Linux `traceroute` uses **UDP** (User Datagram Protocol) by default. It sends UDP packets to high-numbered ports.

No.	Time	Source	Destination	Protocol	Length	Info
23472	934.737220	172.16.4.7	10.7.8.129	ICMP	120	Time-to-live exceeded (Time to live exceeded in transit)
23473	934.778648	10.7.0.5	10.7.8.129	ICMP	70	Redirect (Redirect for host)
23474	934.819482	10.7.0.5	10.7.8.129	ICMP	70	Redirect (Redirect for host)
23475	934.819482	10.7.0.5	10.7.8.129	ICMP	70	Redirect (Redirect for host)
23476	934.820755	10.7.0.5	10.7.8.129	ICMP	70	Redirect (Redirect for host)
23477	934.820755	10.7.0.5	10.7.8.129	ICMP	70	Redirect (Redirect for host)
23478	934.898433	10.7.0.5	10.7.8.129	ICMP	70	Redirect (Redirect for host)
23479	934.899213	10.7.0.5	10.7.8.129	ICMP	70	Redirect (Redirect for host)
23480	934.899213	10.7.0.5	10.7.8.129	ICMP	70	Redirect (Redirect for host)
23481	934.900063	10.7.0.5	10.7.8.129	ICMP	70	Redirect (Redirect for host)
23482	934.980461	10.7.0.5	10.7.8.129	ICMP	70	Redirect (Redirect for host)
23483	934.985273	10.7.8.129	13.89.179.11	TCP	66	49947 → 443 [SYN] Seq=0 Win=65535 Len=0 MSS=1460 WS=256 SACK
23484	934.985720	10.7.0.5	10.7.8.129	ICMP	70	Redirect (Redirect for host)
23485	934.985720	10.7.0.5	10.7.8.129	ICMP	70	Redirect (Redirect for host)
23486	934.985720	10.7.0.5	10.7.8.129	ICMP	70	Redirect (Redirect for host)
23487	935.076176	10.7.0.5	10.7.8.129	ICMP	70	Redirect (Redirect for host)
23488	935.076176	10.7.0.5	10.7.8.129	ICMP	70	Redirect (Redirect for host)
23489	935.076176	10.7.0.5	10.7.8.129	ICMP	70	Redirect (Redirect for host)
23490	935.076176	10.7.0.5	10.7.8.129	ICMP	70	Redirect (Redirect for host)
23491	935.090371	10.7.8.129	155.133.225.20	TLSv1.2	108	Application Data
23492	935.141610	10.7.0.5	10.7.8.129	ICMP	70	Redirect (Redirect for host)
23493	935.141610	10.7.0.5	10.7.8.129	ICMP	70	Redirect (Redirect for host)
23494	935.142950	10.7.0.5	10.7.8.129	ICMP	70	Redirect (Redirect for host)
23495	935.142950	10.7.0.5	10.7.8.129	ICMP	70	Redirect (Redirect for host)
23496	935.178032	155.133.225.20	10.7.8.129	TCP	54	27020 → 53174 [ACK] Seq=1358 Ack=2037 Win=8195 Len=0
23497	935.219262	10.7.0.5	10.7.8.129	ICMP	70	Redirect (Redirect for host)
23498	935.220091	10.7.0.5	10.7.8.129	ICMP	70	Redirect (Redirect for host)

Figure 3: Wireshark capture filtered for ICMP traffic, showing the Echo Requests and Time-to-live exceeded messages generated by `tracert`.

2. **Some hops in your traceroute output may show ***. Provide at least two reasons why a router might not reply.**

   The asterisks (‘***‘) indicate that no reply was received from a router for a given probe. Two primary reasons for this are:

   - **Firewall Configuration:** The router, or a firewall protecting it, is explicitly configured to drop ICMP packets and not send any ICMP error messages. This is a common security practice to prevent network mapping.
   - **ICMP Rate Limiting:** To prevent denial-of-service attacks or reduce load, routers are often configured to limit the rate at which they send ICMP error messages. If probes arrive during a period where this limit is exceeded, the router will simply ignore them.

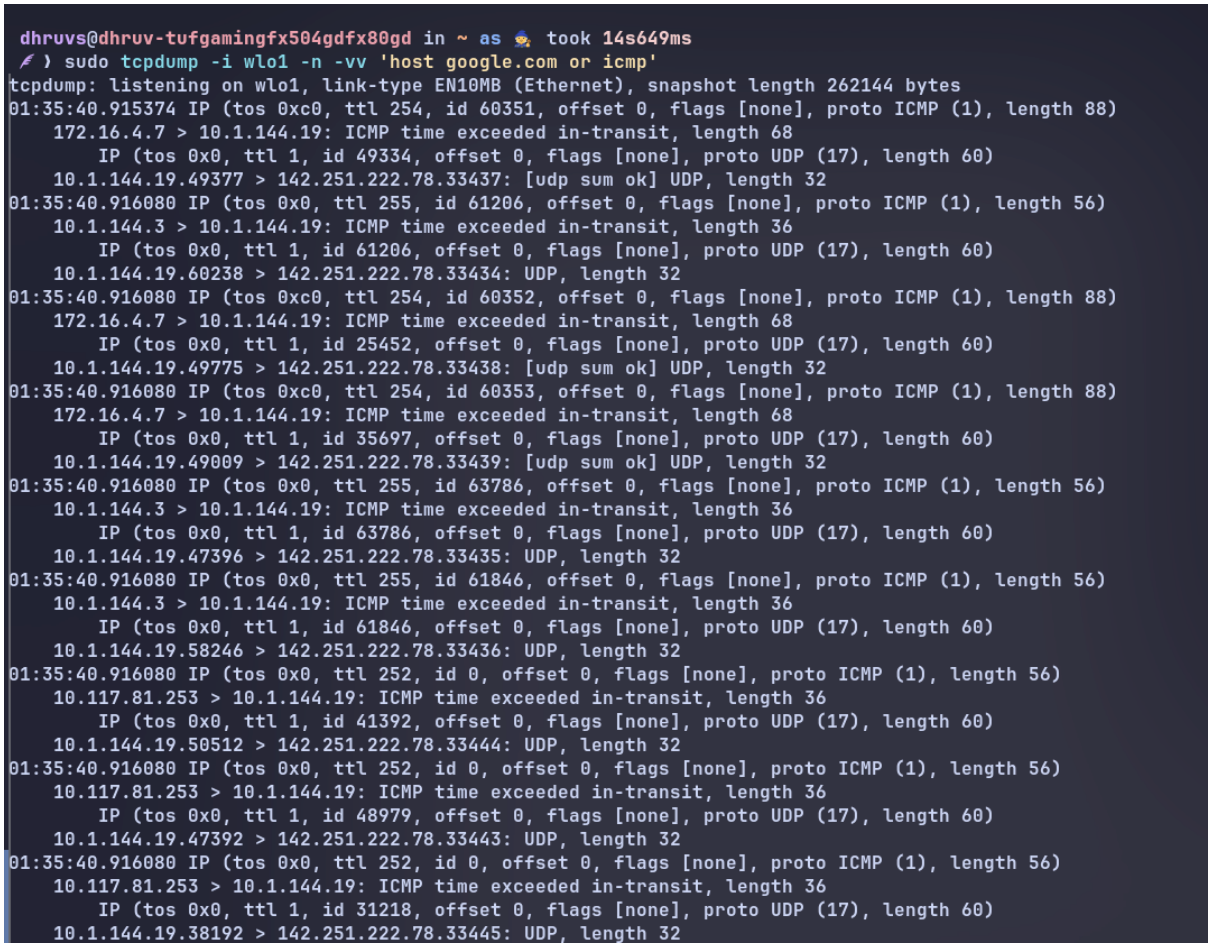

Figure 4: Linux TCPDUMP capture showing the Echo Requests and Time-to-live exceeded messages generated by `traceroute`.

3. **In Linux traceroute, which field in the probe packets changes between successive probes sent to the destination?**

   Linux Analysis

   The analysis was performed by running ‘traceroute google.com‘ in one terminal and ‘sudo tcpdump -i wlo1 -n -vv ’host google.com or icmp’‘ in a second terminal. This allowed for a live capture of both the outgoing probes and the incoming responses.

   Outputs

   Traceroute Output

```
traceroute to google.com (142.251.222.78), 30 hops max, 60 byte packets
1  10.1.144.3 (10.1.144.3)  2.815 ms  2.792 ms  2.781 ms
2  172.16.4.7 (172.16.4.7)  2.062 ms  2.756 ms  2.744 ms
3  14.139.98.1 (14.139.98.1)  3.861 ms  5.453 ms  5.435 ms
4  10.117.81.253 (10.117.81.253)  2.667 ms  2.620 ms  2.596 ms
5  10.154.8.137 (10.154.8.137)  11.950 ms  13.502 ms  12.397 ms
6  10.255.239.170 (10.255.239.170)  15.985 ms  14.107 ms  13.614 ms
7  10.152.7.214 (10.152.7.214)  9.616 ms  9.607 ms  9.597 ms
8  72.14.204.62 (72.14.204.62)  14.436 ms  14.422 ms *
9  * * *
10 192.178.86.200 (192.178.86.200)  16.134 ms 142.251.77.100
    (142.251.77.100)  38.084 ms  38.075 ms
11 192.178.110.208 (192.178.110.208)  10.357 ms 192.178.110.104
    (192.178.110.104)  11.596 ms 192.178.110.110 (192.178.110.110)  38.046
    ms
```

```
12 142.250.208.227 (142.250.208.227)  14.906 ms pnbomb-bp-in-f14.1e100.net
    (142.251.222.78)  10.806 ms 142.250.209.71 (142.250.209.71)  12.880 ms

```

Listing 5: Terminal output of the traceroute command to google.com.

Tcpdump Packet Capture

```
tcpdump: listening on wlo1, link-type EN10MB (Ethernet), snapshot length
    262144 bytes
01:35:40.916080 IP (tos 0x0, ttl 255, id 61206, offset 0, flags [none],
    proto ICMP (1), length 56)
    10.1.144.3 > 10.1.144.19: ICMP time exceeded in-transit, length 36
        IP (tos 0x0, ttl 1, id 61206, offset 0, flags [none], proto UDP
    (17), length 60)
            10.1.144.19.60238 > 142.251.222.78.33434: UDP, length 32
...
01:35:40.929641 IP (tos 0x0, ttl 248, id 63877, offset 0, flags [none],
    proto ICMP (1), length 168)
    10.255.239.170 > 10.1.144.19: ICMP time exceeded in-transit, length 148
        IP (tos 0x0, ttl 2, id 42881, offset 0, flags [none], proto UDP
    (17), length 60)
            10.1.144.19.38871 > 142.251.222.78.33449: [udp sum ok] UDP,
    length 32
...
01:35:41.053762 IP (tos 0x80, ttl 115, id 0, offset 0, flags [none], proto
    ICMP (1), length 56)
    142.251.222.78 > 10.1.144.19: ICMP 142.251.222.78 udp port 33474
    unreachable, length 36
        IP (tos 0x80, ttl 3, id 48364, offset 0, flags [none], proto UDP
    (17), length 60)
            10.1.144.19.49232 > 142.251.222.78.33474: UDP, length 32
...
39 packets captured

```

Listing 6: Abridged tcpdump capture showing UDP probes and ICMP replies.

Analysis and Findings The captured data shows that two fields change for each successive probe:

- **IP Time To Live (TTL):** The TTL field in the IP header is incremented by one for each hop (‘ttl 1‘, ‘ttl 2‘, ‘ttl 3‘, etc.). This is the core mechanism of traceroute.
- **UDP Destination Port:** The destination port in the UDP header is also incremented for each probe (‘...33434‘, ‘...33435‘, etc.). This helps to distinguish between replies and match them to the correct outgoing probe.

4. **At the final hop, how is the response different compared to the intermediate hop?**

- **Intermediate Hops (Both Windows & Linux):** Reply with an ICMP message of Type 11, Code 0, which is **”Time-to-live exceeded.”** This is an error message indicating the packet expired before reaching the destination.
- **Final Hop (Windows `tracert`):** Since the probe is an ICMP Echo Request, the destination replies with a successful ICMP message of Type 0, Code 0, which is an **”Echo Reply.”**

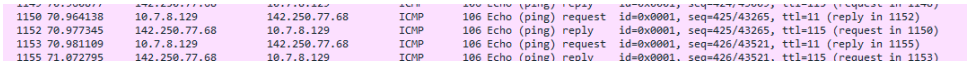

Figure 5: Windows Last Lines

- **Final Hop (Linux `traceroute`):** The UDP probe is sent to a high, unused port. Since no service is listening on that port, the destination's OS sends back an ICMP message of Type 3, Code 3, which is **"Destination Unreachable (Port Unreachable)."** This error message signals to 'traceroute' that the final destination has been reached.

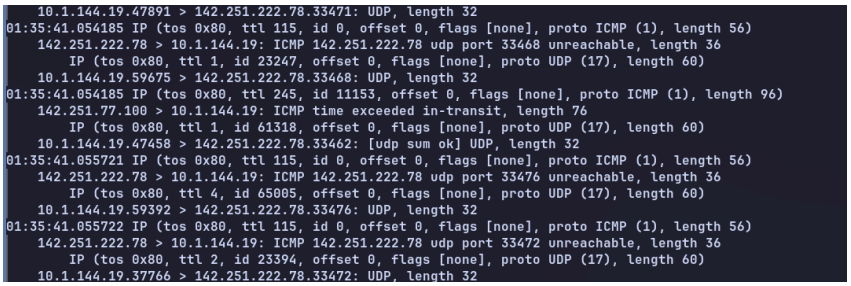

Figure 6: Linux Last Lines

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

   A firewall blocking UDP but allowing ICMP would have different effects on the two systems:

   - **Windows tracert:** Since Windows `tracert` uses ICMP for its probes, it would be **unaffected** by this firewall and would likely complete the trace successfully.
   - **Linux traceroute:** The default Linux `traceroute` uses UDP packets for its probes. This trace would **fail** at the firewall. Its UDP probes would be blocked, and it would not receive any responses from hops beyond that point, resulting in a series of timeouts ('***').

References

1. Traceroute Explanation, available at: `https://share.google/4b7B4sdrua4iLotKd` (Accessed: 15 September 2025).