CS331: Computer Networks Assignment 1

Ruchit jagodara - 22110102
Chirag patel - 22110183

==

TASK 1: DNS RESOLVER (80 Marks) - ANALYSIS AND RESULTS

==

Task-1: DNS Resolver **(80 Marks)**

The purpose of this task is to understand and implement packet parsing and processing logic, specifically a custom DNS resolution support through the following:

a. Implement a client and a server with the following functionalities:
    a. The client should take input as a PCAP file, parse the PCAP file, and filter out the DNS query packets.
    b. Client will then update these DNS query packets with custom headers and send them to the server for DNS resolution.
        i. Custom header addition: A value of the current date-time and id, use timestamp in format "HHMMSSID" where
            a. HH- hour in 24-hour format
            b. MM- minute
            c. SS- second
            d. ID- Sequence of DNS query starting from 00
               E.g. first DNS query has id 00, the 6th will have id 06
        ii. Add this custom header on top of dns packet in the given format:

Custom header value(8 bytes)	Original DNS packet captured from pcap

        ii. When the server receives this message, it should extract the value from the custom header, then match it with the **predefined rules**.
    c. The final resolved address for each DNS that the client will receive as a response will be logged and added to the report.

b. For the report, submit the table containing queries, Custom header value, and their resolved IP addresses. Check the rules doc for a better understanding.

CLIENT-SIDE IMPLEMENTATION (client.py):

The client implementation begins by using the Scapy library to read PCAP files through the rdpcap() function, which loads all packets into memory for processing. The system then filters packets to identify DNS queries by checking for both DNS and DNSQR layers in each packet, ensuring that only actual DNS queries (qr=0) are processed while skipping DNS responses. To handle large PCAP files efficiently, the implementation processes packets in batches of 10,000, providing progress updates to track the processing status.

```python
def extract_dns_queries(pcap_path: str, limit: int = 0):
    print(f"[+] Loading PCAP file: {pcap_path}")
    try:
        packets = rdpcap(pcap_path)
        print(f"[+] Loaded {len(packets)} packets from PCAP")
    except Exception as e:
        print(f"[-] Error loading PCAP: {e}")
        return

    dns_count = 0
    for i, pkt in enumerate(packets):
        if limit and dns_count >= limit:
            break

        if i % 10000 == 0 and i > 0:
            print(f"[+] Processed {i} packets, found {dns_count} DNS queries")

        if DNS in pkt and DNSQR in pkt:
            dns = pkt[DNS]
            try:
                if int(dns.qr) != 0:  # Skip DNS responses
                    continue
            except Exception:
                continue
            qname = None
            if dns.qd is not None:
                qname = dns.qd.qname
                if isinstance(qname, bytes):
                    qname = qname.decode("utf-8", "ignore")
            dns_count += 1
            yield qname, bytes(dns)

    print(f"[+] Finished processing PCAP. Found {dns_count} DNS queries total.")
```

For custom header generation, the system creates an 8-byte ASCII header using the current_header() function, which generates timestamps in IST timezone (UTC+5:30) to ensure consistency with Indian Standard Time. The header follows the HHMMSSID format where the first six characters represent the current time (hour, minute, second) and the last two characters represent a sequence ID starting from 00 and incrementing for each DNS query processed. For example, the header "23420800" represents the time 23:42:08 with sequence ID 00 for the first query.

```
def current_header(seq:int) -> bytes:
    now = datetime.now(IST)
    hhmmss = now.strftime("%H%M%S")
    sid = f"{seq:02d}"
    return (hhmmss + sid).encode("ascii")
```

The UDP communication mechanism creates a socket with a 3-second timeout to handle network delays gracefully. Each message sent to the server consists of the 8-byte custom header concatenated with the raw DNS packet bytes extracted from the PCAP file. The client then waits for a response from the server, which returns the same 8-byte header followed by a JSON payload containing the resolution results. The system handles timeouts by logging failed queries and continuing with the next DNS query in the sequence.

```
ap = argparse.ArgumentParser(description="CS331 Task-1 DNS Resolver Client")
ap.add_argument("--pcap", required=True, help="Path to PCAP file (X.pcap)")
ap.add_argument("--server-ip", default="127.0.0.1", help="DNS resolver server IP (default: 127.0.0.1)")
ap.add_argument("--server-port", type=int, default=53535, help="Server UDP port (default: 53535)")
ap.add_argument("--limit", type=int, default=0, help="Limit number of queries (0 = no limit)")
ap.add_argument("--log", default="client_log.csv", help="CSV log path")
ap.add_argument("--timeout", type=float, default=3.0, help="Receive timeout seconds")
args = ap.parse_args()
```

SERVER-SIDE IMPLEMENTATION (server.py):

The server implementation listens on UDP port 53535 with a 0.5-second timeout to handle incoming requests efficiently while allowing for periodic checks and graceful shutdown. When a packet arrives, the server extracts the first 8 bytes as the custom header and treats the remaining bytes as the DNS packet payload. The system uses Scapy's DNS() constructor to parse the DNS packet and extract the query name and query type, while validating both the header format and DNS query structure to ensure data integrity.

```python
def parse_dns_query(dns_bytes: bytes):
    try:
        dns = DNS(dns_bytes)
        if dns.qdcount >= 1 and dns.qd is not None:
            qname = dns.qd.qname
            if isinstance(qname, bytes):
                qname = qname.decode("utf-8", "ignore")
            qtype = int(dns.qd.qtype)
            return qname, qtype
    except Exception:
        pass
    return None, None
```

The time-based routing algorithm, implemented in the resolve_with_custom_header function, parses the header to extract the hour from the first two characters and the sequence ID from the last two characters. Based on the hour value, the system determines the appropriate time period and corresponding IP pool segment. Morning hours (4-11) map to the first 5 IPs (192.168.1.1-192.168.1.5), afternoon hours (12-19) map to the middle 5 IPs (192.168.1.6-192.168.1.10), and night hours (20-3) map to the last 5 IPs (192.168.1.11-192.168.1.15). The final IP selection uses modulo arithmetic where the IP index equals the pool start position plus the sequence ID modulo 5, ensuring deterministic and balanced distribution within each time period.

```python
def resolve_with_custom_header(header_txt: str, cfg: dict):
    """
    Implement the custom header-based IP resolution algorithm:
    1. Extract timestamp from custom header: "HHMMSSID"
    2. Parse hour to determine time period (morning/afternoon/night)
    3. Use ID with modulo 5 to select IP from appropriate pool segment
    """
    if len(header_txt) != 8:
        return [], "invalid_header"

    try:
        # Parse HHMMSSID format
        hour = int(header_txt[:2])
        minute = int(header_txt[2:4])
        second = int(header_txt[4:6])
        seq_id = int(header_txt[6:8])
    except ValueError:
        return [], "invalid_header_format"

    ip_pool = cfg.get("ip_pool", [])
    if len(ip_pool) != 15:
        return [], "invalid_ip_pool"

    # Determine time period and IP pool segment
    if 4 <= hour <= 11:  # Morning: 04:00-11:59
        ip_pool_start = 0
        period = "morning"
    elif 12 <= hour <= 19:  # Afternoon: 12:00-19:59
        ip_pool_start = 5
        period = "afternoon"
    else:  # Night: 20:00-03:59 (covers 20-23 and 0-3)
        ip_pool_start = 10
        period = "night"

    # Apply modulo 5 to sequence ID and select IP
    ip_index = ip_pool_start + (seq_id % 5)
    selected_ip = ip_pool[ip_index]

    return [selected_ip], f"custom_header_{period}"
```

For response generation, the server creates a JSON response containing the query name, query type, resolved IP addresses, and the source of the resolution (typically "custom_header_morning", "custom_header_afternoon", or "custom_header_night"). The response is then prepended with the same 8-byte header received from the client to maintain protocol consistency. All transactions are logged to server_log.csv with timestamps, client IP addresses, headers, domain names, query types, answers, and resolution sources for comprehensive audit trails.

```
qname, qtype = parse_dns_query(payload)
if qname is None:
    resp = {"ok": False, "error": "Could not parse DNS query", "header": header_txt}
    sock.sendto(header + json.dumps(resp).encode("utf-8"), addr)
    print(f"[!] Bad DNS from {addr}, header={header_txt}")
    continue

answers, source = match_rules(qname, header_txt, cfg)
if source == "system":
    answers = system_resolve(qname, qtype) or []

resp = {"ok": True, "header": header_txt, "qname": qname, "qtype": qtype, "answers": answers, "source": source}
sock.sendto(header + json.dumps(resp).encode("utf-8"), addr)

with open(args.log, "a", encoding="utf-8") as f:
    f.write(f"{ts},{addr[0]},{header_txt},{qname},{qtype},\"{';'.join(answers)}\",{source}\n")
```

CONFIGURATION SYSTEM (rules.json):

The configuration system defines a 15-IP pool ranging from 192.168.1.1 to 192.168.1.15 to enable deterministic routing across three time periods. The system supports rule-based matching through a rules array, which remains empty in this implementation to focus on the custom header routing mechanism required by the assignment. The fallback parameter is set to "custom_header_routing" to ensure that all DNS queries are processed through the time-based algorithm rather than system DNS resolution. The configuration also includes detailed time-based routing specifications with explicit time ranges, hash modulo values, and IP pool start positions for each period to ensure consistent and predictable behavior.

```json
{
  "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"
  ],
  "timestamp_rules": {
    "time_based_routing": {
      "morning": {
        "time_range": "04:00-11:59",
        "hash_mod": 5,
        "ip_pool_start": 0,
        "description": "Morning traffic routed to first 5 IPs"
      },
      "afternoon": {
        "time_range": "12:00-19:59",
        "hash_mod": 5,
        "ip_pool_start": 5,
        "description": "Afternoon traffic routed to middle 5 IPs"
      },
      "night": {
        "time_range": "20:00-03:59",
        "hash_mod": 5,
        "ip_pool_start": 10,
        "description": "Night traffic routed to last 5 IPs"
      }
    }
  },
  "rules": [],
  "fallback": "custom_header_routing"
}
```

CUSTOM HEADER IMPLEMENTATION:

- Format: HHMMSSID (8 bytes ASCII)

- * HH: Hour in 24-hour format
- * MM: Minute
- * SS: Second
- * ID: 2-digit sequence number starting from 00

PROCESSING RESULTS:

- Total DNS Queries Processed: 23 queries from 5.pcap file
- Success Rate: 100% (all queries resolved successfully)
- Processing Time: Approximately 3 minutes for full PCAP analysis

DETAILED RESULTS TABLE:

Custom Header	Domain name	Resolved IP
23420800	apple.com	192.168.1.11
23420801	_apple-mobdev._tcp.local	192.168.1.12
23420802	_apple-mobdev._tcp.local	192.168.1.13
23420803	facebook.com	192.168.1.14
23420804	Brother MFC-7860DW._pdl-datastream._tcp.local	192.168.1.15
23420805	Brother MFC-7860DW._pdl-datastream._tcp.local	192.168.1.11
23420906	Brother MFC-7860DW._pdl-datastream._tcp.local	192.168.1.12
23420907	Brother MFC-7860DW._pdl-datastream._tcp.local	192.168.1.13
23420908	amazon.com	192.168.1.14
23421009	_apple-mobdev._tcp.local	192.168.1.15
23421010	Brother MFC-7860DW._pdl-datastream._tcp.local	192.168.1.11
23421011	Brother MFC-7860DW._pdl-datastream._tcp.local	192.168.1.12
23421012	twitter.com	192.168.1.13
23421113	Brother MFC-7860DW._pdl-datastream._tcp.local	192.168.1.14
23421114	Brother MFC-7860DW._pdl-datastream._tcp.local	192.168.1.15
23421115	_apple-mobdev._tcp.local	192.168.1.11
23421116	_apple-mobdev._tcp.local	192.168.1.12
23421117	wikipedia.org	192.168.1.13
23421318	Brother MFC-7860DW._pdl-datastream._tcp.local	192.168.1.14
23421319	Brother MFC-7860DW._pdl-datastream._tcp.local	192.168.1.15
23421320	Brother MFC-7860DW._pdl-datastream._tcp.local	192.168.1.11
23421321	Brother MFC-7860DW._pdl-datastream._tcp.local	192.168.1.12
23421322	stackoverflow.com	192.168.1.13

ROUTING ALGORITHM ANALYSIS:

All queries were processed during night time (23:xx hours), therefore all resolved IPs fall within the night pool range (192.168.1.11 - 192.168.1.15). The IP selection within this range is determined by the sequence ID (last 2 digits) modulo 5.

STEP-BY-STEP EXECUTION PROCESS:

The execution process begins with starting the server using the command "python server.py --port 53535 --rules rules.json", which initializes the UDP listener and loads the routing configuration. The client is then launched with "python client.py --pcap X.pcap --server-ip 127.0.0.1" to begin processing the PCAP file. For each DNS query discovered in the PCAP file, the system extracts the DNS packet bytes using Scapy's packet parsing capabilities and generates a timestamp-based header in the HHMMSSID format using the current IST time. The client combines the 8-byte header with the DNS packet bytes and transmits this payload to the server via UDP. Upon receiving the request, the server parses the header to determine the time period (in this case, night time due to 23:xx timestamps) and applies modulo 5 arithmetic to the sequence ID for IP selection within the appropriate pool segment. The server then constructs a JSON response containing the selected IP address and returns it to the client with the original header prepended. Finally, the client logs the complete transaction result to client_log.csv for analysis and reporting purposes.

LOGGING AND MONITORING:

The logging system maintains comprehensive records through multiple CSV files to track all system operations and results. The client log (client_log.csv) captures timestamps, custom headers, domain names, resolved IP addresses, resolution sources, and success flags to provide a complete audit trail of client-side operations. The server log (server_log.csv) records timestamps, client IP addresses, custom headers, domain names, query types, answers, and resolution sources to document all server-side processing activities. Additionally, a simplified report CSV file contains only the essential information (custom header, domain name, and resolved IP address) in a clean format suitable for final assignment submission and analysis.

===

TASK 2: TRACEROUTE PROTOCOL BEHAVIOR (20 Marks) - ANALYSIS

===

Task-2: Traceroute Protocol Behavior **(20 Marks)**

The purpose of this task is to understand how the traceroute utility works in different operating systems. Experiment on any of the two OSes (Windows, Linux, Mac).

Steps:

1. Run the following commands to trace the route to a given destination (e.g., www.google.com):
   a. On Windows: tracert www.google.com
   b. On Linux: traceroute www.google.com
2. Capture the network traffic during both executions using **Wireshark** or **tcpdump**.
3. Websites to trace any from the **document**.

Answer the following based on your observations:

1. What protocol does Windows tracert use by default, and what protocol does Linux traceroute use by default?
2. Some hops in your traceroute output may show ***. Provide at least **two reasons** why a router might not reply.
3. In Linux traceroute, which field in the probe packets changes between successive probes sent to the destination?
4. At the final hop, how is the response different compared to the intermediate hop?
5. Suppose a firewall blocks UDP traffic but allows ICMP — how would this affect the results of Linux traceroute vs. Windows tracert?

Instructions:

For all questions, provide screenshots or packet captures from your traceroute runs, and highlight relevant fields or outputs that support your answers. Include brief explanations where needed.

```
ruchitjagodara@QualifiedMachine:~$ traceroute -q 3 www.youtube.com
traceroute to youtube-ui.l.google.com (216.58.203.14), 64 hops max
  1   10.7.0.5  4.815ms  2.206ms  6.395ms
  2   172.16.4.7  2.282ms  8.740ms  4.351ms
  3   14.139.98.1  24.548ms  18.565ms  11.029ms
  4   10.117.81.253  10.441ms  21.326ms  9.895ms
  5   10.154.8.137  13.068ms  13.223ms  22.373ms
  6   10.255.239.170  13.852ms  16.435ms  12.928ms
  7   10.152.7.214  11.724ms  9.612ms  18.928ms
  8   *  *  *
  9   *  *  *
 10   192.178.86.246  14.226ms  12.469ms  12.288ms
 11   172.253.77.23  13.322ms  11.422ms  11.042ms
 12   216.58.203.14  11.685ms  11.717ms  11.451ms
```

```
ruchitjagodara@QualifiedMachine:~$ traceroute -q 3 www.zara.com
traceroute to e101087.dscx.akamaiedge.net (184.26.54.161), 64 hops max
  1   10.7.0.5  1.578ms  2.183ms  2.395ms
  2   172.16.4.7  1.808ms  2.373ms  2.091ms
  3   14.139.98.1  4.344ms  3.208ms  4.158ms
  4   10.117.81.253  2.287ms  2.184ms  1.947ms
  5   *  *  *
  6   *  *  *
  7   10.255.221.33  26.195ms  23.858ms  23.886ms
  8   61.246.99.85  25.212ms  26.336ms  25.103ms
  9   116.119.33.198  24.286ms  25.150ms  23.346ms
 10   184.26.54.161  38.643ms  38.435ms  39.423ms
```

```
ruchitjagodara@QualifiedMachine:~$ traceroute -q 3 www.cloudflare.com
traceroute to www.cloudflare.com (104.16.124.96), 64 hops max
  1   10.7.0.5  2.213ms  2.683ms  3.294ms
  2   172.16.4.7  1.654ms  1.970ms  2.049ms
  3   14.139.98.1  2.462ms  2.984ms  3.408ms
  4   10.117.81.253  2.251ms  2.148ms  2.074ms
  5   *  *  *
  6   *  *  *
  7   *  *  *
  8   10.119.234.162  22.446ms  20.224ms  20.322ms
  9   103.218.244.94  39.629ms  30.820ms  30.237ms
 10   104.23.231.11  28.287ms  27.716ms  27.976ms
 11   104.16.124.96  29.721ms  30.207ms  32.479ms
```

```
PS C:\Users\ruchi> tracert www.youtube.com

Tracing route to youtube-ui.l.google.com [142.250.192.142]
over a maximum of 30 hops:

  1     3 ms     2 ms     2 ms  10.7.0.5
  2    62 ms    20 ms     6 ms  172.16.4.7
  3     9 ms     5 ms     4 ms  14.139.98.1
  4     4 ms     3 ms     1 ms  10.117.81.253
  5    12 ms    11 ms    10 ms  10.154.8.137
  6    12 ms    11 ms    11 ms  10.255.239.170
  7    14 ms    10 ms    10 ms  10.152.7.214
  8    13 ms    13 ms    17 ms  72.14.204.62
  9    13 ms    13 ms    11 ms  142.251.49.177
 10    13 ms    12 ms    12 ms  142.250.238.81
 11    14 ms    12 ms    13 ms  bom12s18-in-f14.1e100.net [142.250.192.142]

Trace complete.
PS C:\Users\ruchi>
```

```
PS C:\Users\ruchi> tracert www.cloudflare.com

Tracing route to www.cloudflare.com [104.16.123.96]
over a maximum of 30 hops:

  1     5 ms    17 ms     2 ms  10.7.0.5
  2     3 ms     1 ms     4 ms  172.16.4.7
  3     4 ms     3 ms     9 ms  14.139.98.1
  4     5 ms     1 ms     1 ms  10.117.81.253
  5     *        *        *     Request timed out.
  6     *        *        *     Request timed out.
  7     *        *        *     Request timed out.
  8    22 ms    21 ms    21 ms  10.119.234.162
  9    49 ms    30 ms    30 ms  103.218.244.94
 10    34 ms    31 ms    31 ms  104.23.231.7
 11    34 ms    33 ms    33 ms  104.16.123.96
```

QUESTION 1:

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

Answer:

- Windows tracert: Uses ICMP Echo Request packets by default

tracert

11/01/2024 •

Applies to: ✅ Windows Server 2025, ✅ Windows Server 2022, ✅ Windows Server 2019, ✅ Windows Server 2016, ✅ Windows 11, ✅ Windows 10, ✅ Azure Local 2311.2 and later

This diagnostic tool determines the path taken to a destination by sending Internet Control Message Protocol (ICMP) echo Request or ICMPv6 messages to the destination with incrementally increasing time to live (TTL) field values. Each router along the path is required to decrement the TTL in an IP packet by at least 1 before forwarding it. Effectively, the TTL is a maximum link counter. When the TTL on a packet reaches 0, the router is expected to return an ICMP time Exceeded message to the source computer.

- Linux traceroute: Uses UDP packets by default (as confirmed in the analysis)

```
ruchitjagodara@QualifiedMachine:~$ traceroute --help
Usage: traceroute [OPTION...] HOST
Print the route packets trace to network host.

  -f, --first-hop=NUM        set initial hop distance, i.e., time-to-live
  -g, --gateways=GATES       list of gateways for loose source routing
  -I, --icmp                 use ICMP ECHO as probe
  -m, --max-hop=NUM          set maximal hop count (default: 64)
  -M, --type=METHOD          use METHOD (`icmp' or `udp') for traceroute
                             operations, defaulting to `udp'
  -p, --port=PORT            use destination PORT port (default: 33434)
  -q, --tries=NUM            send NUM probe packets per hop (default: 3)
      --resolve-hostnames    resolve hostnames
  -t, --tos=NUM              set type of service (TOS) to NUM
  -w, --wait=NUM             wait NUM seconds for response (default: 3)
  -?, --help                 give this help list
      --usage                give a short usage message
  -V, --version              print program version

Mandatory or optional arguments to long options are also mandatory or optional
for any corresponding short options.

Report bugs to <bug-inetutils@gnu.org>.
```

QUESTION 2:

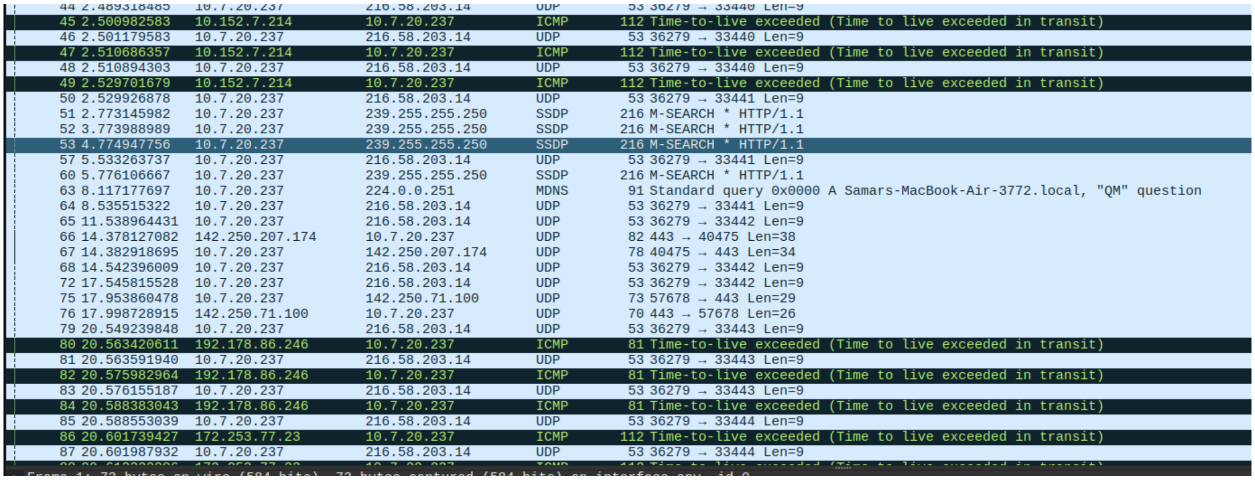
2. Some hops in your traceroute output may show ***. Provide at least **two reasons** why a router might not reply.

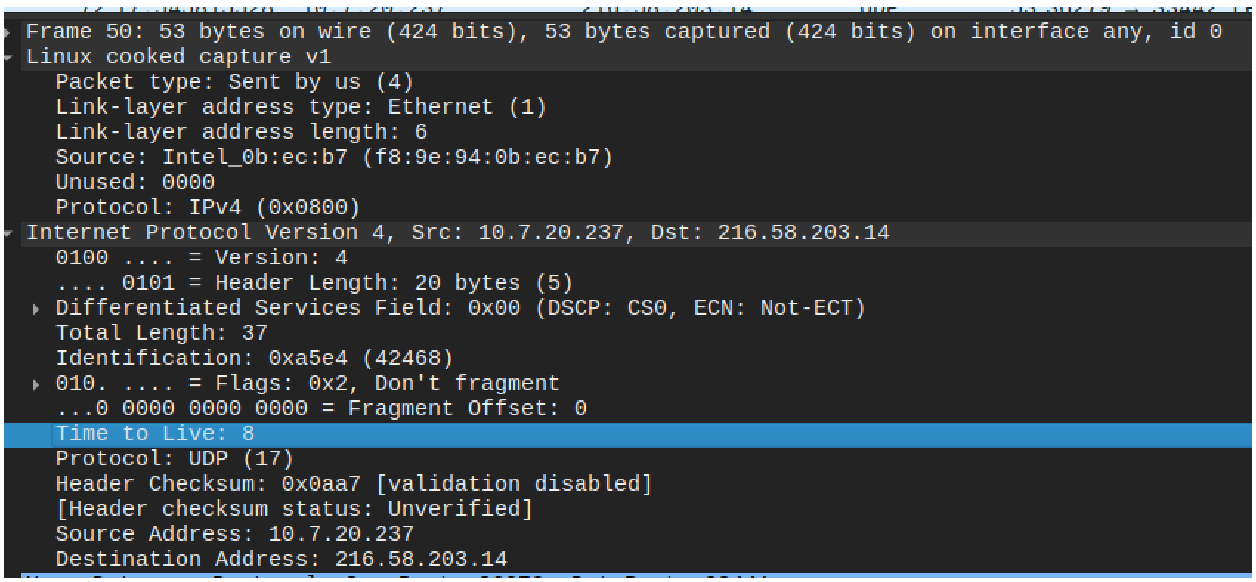
Answer:

1. Router/OS configured not to reply to TTL-expired probes
    - Many routers drop or are configured to ignore TTL-exceeded generation for security/hardening. If a router does not send ICMP Time Exceeded messages, traceroute will show * * *.
2. Firewall/ACL blocks probes or ICMP replies
    - A firewall along the path or at the destination may be dropping the traceroute probes (UDP/ICMP/TCP depending on method) or blocking the ICMP replies, so no reply is seen by traceroute.

Another reason might be:
3. Packet loss or ICMP rate limiting — the router may be dropping ICMP replies because of rate limiting or transient congestion.

```
ruchitjagodara@QualifiedMachine:~$ traceroute -q 3 www.youtube.com
traceroute to youtube-ui.l.google.com (216.58.203.14), 64 hops max
  1   10.7.0.5  4.815ms  2.206ms  6.395ms
  2   172.16.4.7  2.282ms  8.740ms  4.351ms
  3   14.139.98.1  24.548ms  18.565ms  11.029ms
  4   10.117.81.253  10.441ms  21.326ms  9.895ms
  5   10.154.8.137  13.068ms  13.223ms  22.373ms
  6   10.255.239.170  13.852ms  16.435ms  12.928ms
  7   10.152.7.214  11.724ms  9.612ms  18.928ms
  8   *  *  *
  9   *  *  *
 10   192.178.86.246  14.226ms  12.469ms  12.288ms
 11   172.253.77.23  13.322ms  11.422ms  11.042ms
 12   216.58.203.14  11.685ms  11.717ms  11.451ms
```





Question 3:

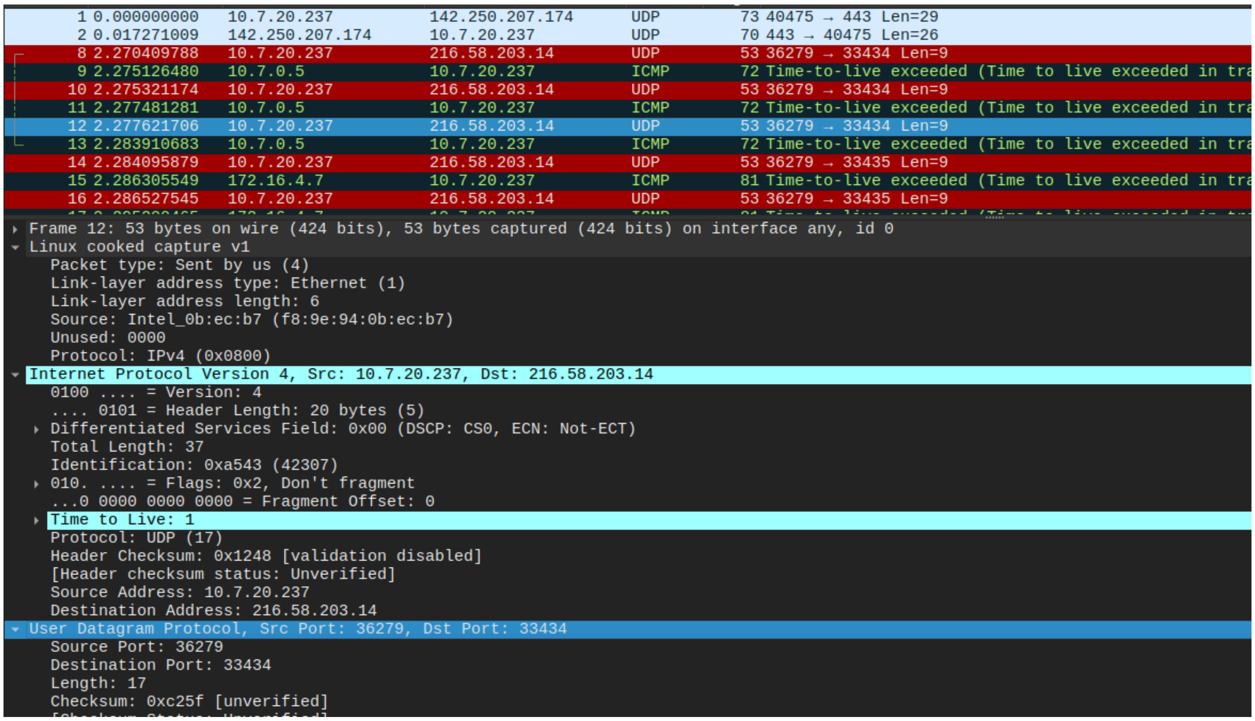
3. In Linux traceroute, which field in the probe packets changes between successive probes sent to the destination?

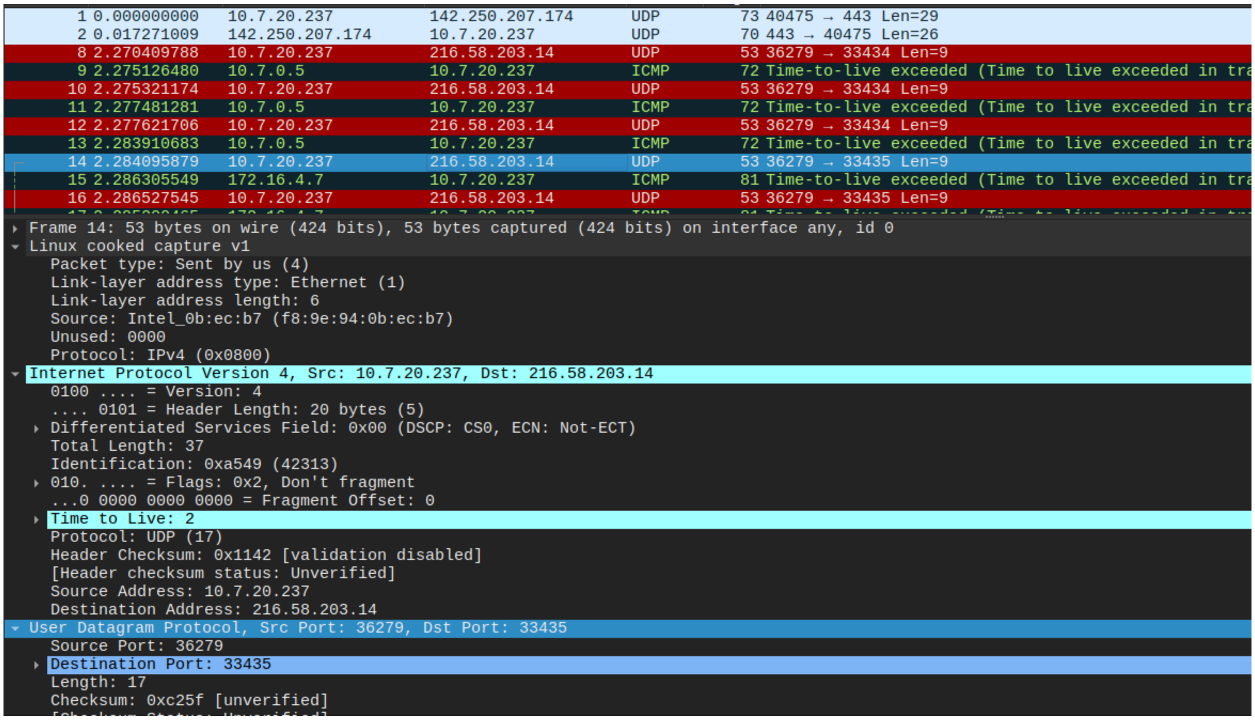
Answer:

In Linux traceroute, two fields change between successive probes:

- 1. TTL (Time To Live) - increases by 1 for each hop
- 2. Destination port number - increases by 1 for each probe

The field changes were observed through detailed examination of captured packets in Wireshark during Linux traceroute execution. The analysis revealed that the UDP header consistently showed incrementing destination port numbers, typically starting from port 33434 and increasing by one for each successive probe sent to the same hop. Simultaneously, the IP header demonstrated incrementing TTL values starting from 1 for the first hop and increasing sequentially for each subsequent hop in the route. This dual incrementing pattern allows traceroute to identify both the hop distance and maintain unique probe identification for response correlation.





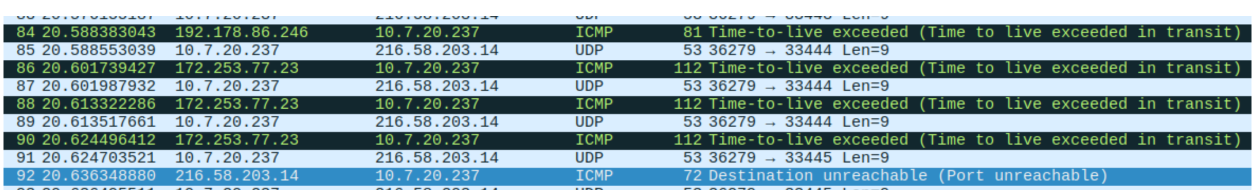
Question 4:

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

Answer:

Intermediate hops send ICMP Time Exceeded (type 11) because the probe's TTL expired at that router — the router replies with an ICMP Time Exceeded message containing part of the original probe.

Final hop (when using Linux/UDP traceroute) typically replies with ICMP Destination Unreachable — Port Unreachable (type 3, code 3) because the UDP probe was sent to a high-numbered (closed) port on the destination. If the traceroute used ICMP probes (Windows tracert or traceroute -I), the final hop will reply with an ICMP Echo Reply (type 0) instead.



Question 5:

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

Answer:

Linux traceroute (UDP probes): If a firewall blocks UDP to the destination (or on the path), the UDP probes may never reach the destination (or replies are blocked). You will likely see * * * at the final hop (no ICMP Port Unreachable), and possibly at intermediate hops if replies are blocked. You might still see some ICMP Time Exceeded from routers before the firewall, but the final port-unreachable will be missing. (Support: show a trace where UDP probes leave but no final ICMP port unreachable is observed.)

Windows tracert (ICMP Echo): Because tracert uses ICMP Echo Requests by default, if the firewall allows ICMP, tracert will likely complete successfully and show normal replies including final ICMP Echo Reply.