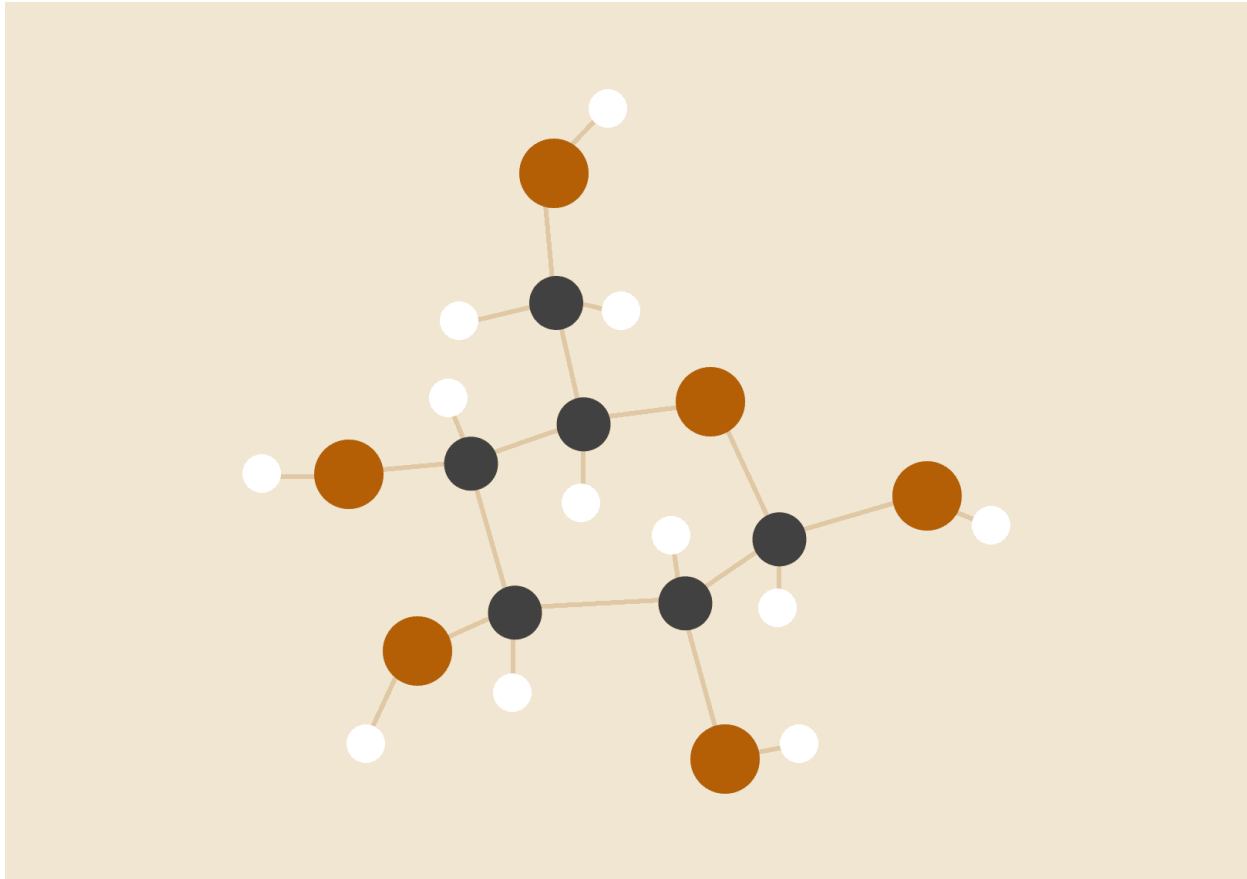


ASSIGNMENT - 1

CS 331: Computer Networks Fall 2025



Devansh Lodha - 23110091
&
Mohit Kamlesh Panchal - 23110208

15.09.2025

1. TASK 1: DNS RESOLVER

1.1. INTRODUCTION

The primary objective of this task was to design and implement a custom client-server application for DNS resolution, demonstrating a comprehensive understanding of network packet parsing, protocol design, and socket programming. The system's core function is to parse DNS queries from a provided PCAP file, augment each query with a custom 8-byte application-layer header, and resolve these queries using a server that implements a set of time-based load-balancing rules. The implementation successfully utilizes both low-level raw sockets for manual header construction and standard datagram sockets for a high-level, portable approach.

1.2. THEORETICAL FOUNDATIONS AND DESIGN PRINCIPLES

The implementation is grounded in the client-server paradigm of the Application Layer (Layer 7) of the OSI model. A bespoke protocol, defined by the custom HHMMSSID header, is layered on top of the User Datagram Protocol (UDP). UDP was selected as the transport protocol, aligning with standard DNS behavior (RFC 1035), which prioritizes low-latency, connectionless communication over the guaranteed delivery offered by TCP.

A key part of the implementation was exploring two distinct networking methodologies:

- **Raw Sockets (socket.SOCK_RAW):** This approach provides direct access to the Network Layer (Layer 3), enabling the manual construction of complete IPv4 packets. This requires strict adherence to the packet structure defined in IETF RFC 791. The implementation correctly uses the IP_HDRINCL socket option and manually calculates the IP header checksum. For maximum compatibility, the UDP checksum field was set to zero, delegating its calculation to the operating system's kernel, a standard and robust practice.
- **Datagram Sockets (socket.SOCK_DGRAM):** This is the standard, high-level abstraction for UDP communication. It delegates all Layer 3 and 4 header construction and checksum calculations to the OS networking stack, providing a portable and reliable interface.

mDNS: Multicast DNS (mDNS), defined in RFC 6762, is a zero-configuration protocol for resolving hostnames to IP addresses in local networks without a traditional DNS server. Operating over UDP on port 5353 with multicast address 224.0.0.251 (or ff02::fb for IPv6), mDNS enables peer-to-peer name resolution and service discovery, typically for domains ending in .local (e.g., _apple-mobdev._tcp.local). Unlike traditional DNS, which uses unicast queries to a server on port 53, mDNS broadcasts queries to all local devices, supporting protocols like Bonjour or Avahi for discovering services such as printers or Apple devices. Its decentralized nature and reliance on multicast make it ideal for small, ad-hoc networks but distinct from the hierarchical, server-based traditional DNS system. Notably, Wireshark's default DNS query filter excludes mDNS due to its port 53 restriction, but our implementation's initial analysis without this condition revealed mDNS traffic, highlighting its presence in the pcap file.

1.3. PCAP FILE ANALYSIS

A preliminary analysis of the provided 9.pcap file (file size: 377MB) was conducted to understand the characteristics of the input data. This analysis informs the design of the processing logic and provides context for the final results.

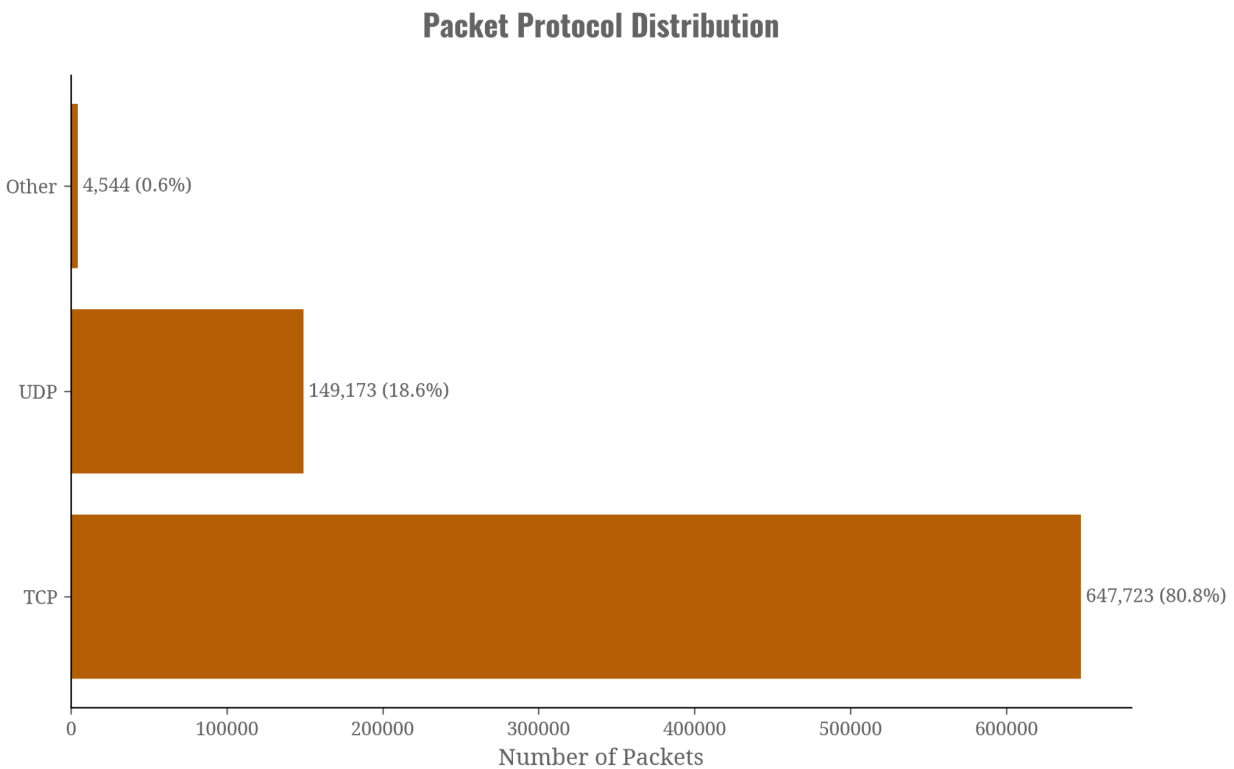


Figure 1: Packet Protocol Distribution.

The analysis confirms that TCP (80.8%) and UDP (18.6%) packets are the dominant protocols in the capture. This distribution is consistent with typical internet traffic, where TCP is used for reliable, stream-based communication (e.g., HTTP, FTP) and UDP is used for latency-sensitive, query-response protocols like DNS. The project correctly focuses on filtering the UDP packets containing these DNS queries.

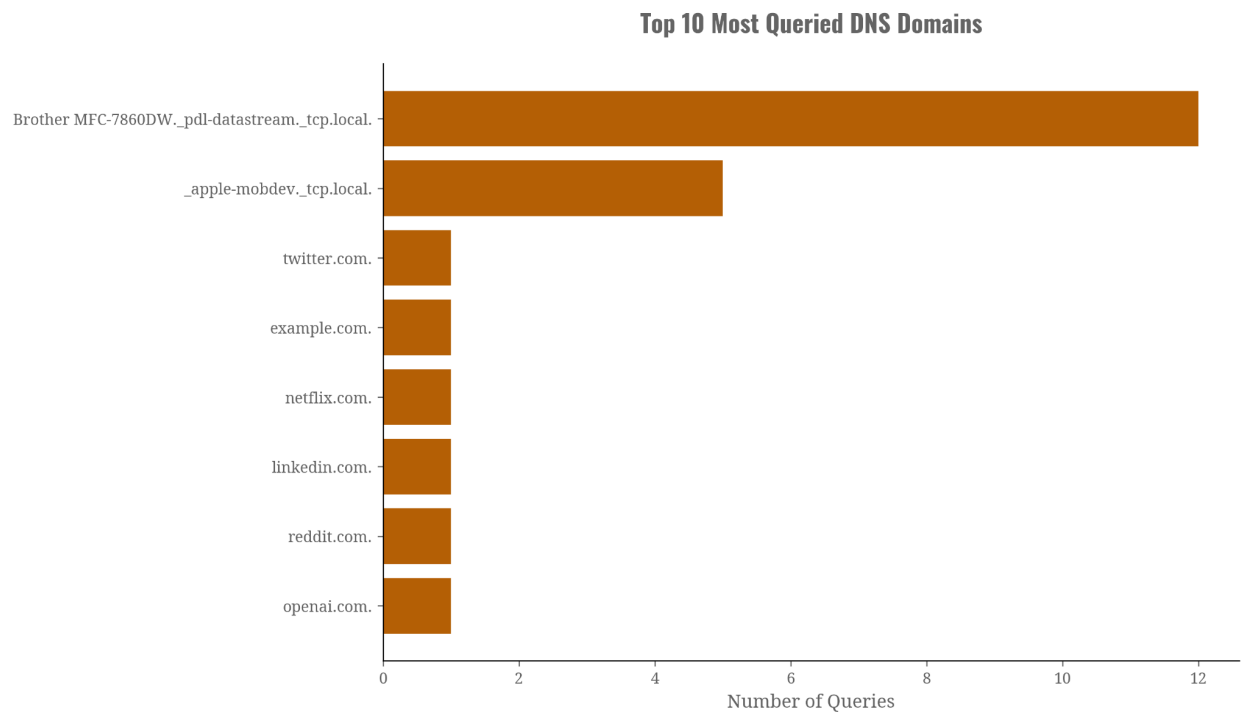


Figure 2: Top 10 Most Queried DNS Domains.

The data shows a mix of queries for public internet domains (e.g., twitter.com, netflix.com) and local network service discovery names (e.g., _apple-mobdev._tcp.local.). This indicates the traffic capture likely originated from a standard consumer or small office network environment where devices automatically discover services like printers and other Apple devices.

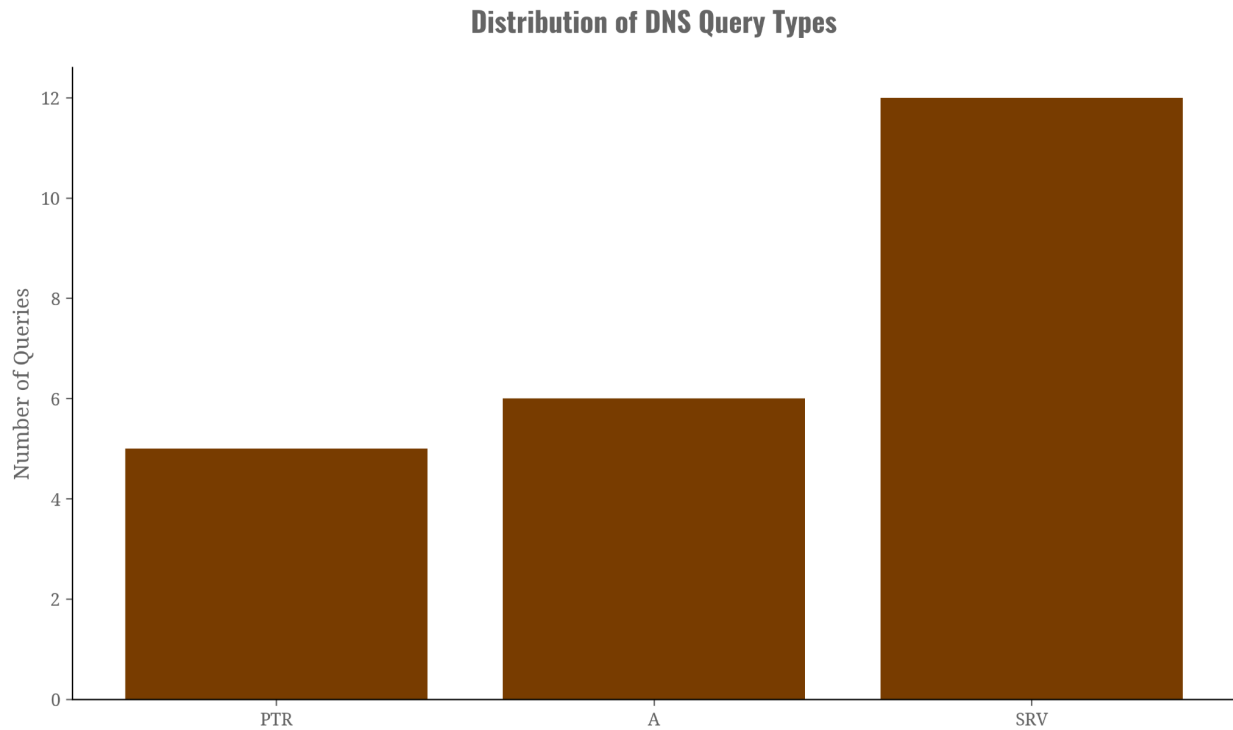


Figure 3: Distribution of DNS Query Types.

The queries are predominantly for 'A' (IPv4) records, which is the expected behavior for a client system performing lookups to establish connections to web services. The presence of 'PTR' queries suggests reverse DNS lookups, often used for network diagnostics or security checks, while 'SRV' records are used for service discovery. This variety confirms the need for a robust parsing mechanism capable of handling standard query types.

1.4. METHODOLOGY AND IMPLEMENTATION

The application was developed in Python 3, utilizing the scapy library for reliable DNS packet parsing and matplotlib for data analysis. A Makefile automates the setup and execution workflows.

CLIENT ARCHITECTURE

The client employs an efficient three-phase "Scan-Collect-Send-Receive" architecture:

1. Scan: It performs a single pass over the PCAP file to collect all DNS query payloads into an in-memory list, preserving their original timestamps.
2. Send: It iterates through the queries, constructs the custom HHMMSSID header for each (with the sequence ID correctly starting from 00 as per the specification), and

sends them to the server in a rapid burst.

3. Receive: It enters a dedicated loop to collect responses. Crucially, the client parses an echoed header from each response to correctly correlate it with the original request, overcoming the unordered nature of UDP.

SERVER LOGIC AND CUSTOM PROTOCOL

The server listens for incoming packets and performs two essential filtering steps: it first ensures the packet is destined for its specific port and then validates the packet structure. This ensures stability by preventing crashes from stray network traffic. For valid requests, it executes the IP selection algorithm:

1. It parses the HH (hour) and ID values from the custom header.
2. It uses the hour to select the appropriate IP pool (morning, afternoon, or night) and its corresponding rules.
3. It calculates the final IP address using the pool's starting index and the result of the $ID \% \text{hash_mod}$ operation.
4. To ensure correct client-side correlation, the server constructs a response by prepending the original custom header to the resolved IP address string before sending it back.

1.5. RESULTS

The application was executed successfully, processing all 23 DNS queries from the 9.pcap file and receiving 23 corresponding resolved IP addresses. The final resolution table matches the expected output based on the assignment's time-based rules.

Table 1: DNS (including mDNS) queries

CUSTOM HEADER (HHMMSSID)	DOMAIN NAME	RESOLVED IP ADDRESS
03324401	_apple-mobdev._tcp.local.	192.168.1.12
03324402	_apple-mobdev._tcp.local.	192.168.1.13
18041603	twitter.com.	192.168.1.9
03332404	Brother MFC-7860DW._pdl-datastream._tcp.local.	192.168.1.15

03332505	Brother MFC-7860DW._pdl-datastream._tcp.local.	192.168.1.11
18041606	example.com.	192.168.1.7
18041607	netflix.com.	192.168.1.8
03342508	Brother MFC-7860DW._pdl-datastream._tcp.local.	192.168.1.14
03342609	Brother MFC-7860DW._pdl-datastream._tcp.local.	192.168.1.15
18041610	linkedin.com.	192.168.1.6
03352011	_apple-mobdev._tcp.local.	192.168.1.12
03352512	Brother MFC-7860DW._pdl-datastream._tcp.local.	192.168.1.13
03352613	Brother MFC-7860DW._pdl-datastream._tcp.local.	192.168.1.14
18041614	reddit.com.	192.168.1.10
03362515	Brother MFC-7860DW._pdl-datastream._tcp.local.	192.168.1.11
03362616	Brother MFC-7860DW._pdl-datastream._tcp.local.	192.168.1.12
03363617	_apple-mobdev._tcp.local.	192.168.1.13
03363618	_apple-mobdev._tcp.local.	192.168.1.14
18041619	openai.com.	192.168.1.10
03372120	Brother MFC-7860DW._pdl-datastream._tcp.local.	192.168.1.11
03372221	Brother MFC-7860DW._pdl-datastream._tcp.local.	192.168.1.12

03372622	Brother MFC-7860DW._pdl-datastream._tcp.local.	192.168.1.13
03372723	Brother MFC-7860DW._pdl-datastream._tcp.local.	192.168.1.14

Table 2: DNS queries with port = 53 check

CUSTOM HEADER (HHMMSSID)	DOMAIN NAME	RESOLVED IP ADDRESS
18041603	twitter.com.	192.168.1.9
18041606	example.com.	192.168.1.7
18041607	netflix.com.	192.168.1.8
18041610	linkedin.com.	192.168.1.6
18041614	reddit.com.	192.168.1.10
18041619	openai.com.	192.168.1.10

VERIFICATION

To verify the correctness of the server's resolution logic, we can manually check two examples from the table against the defined rules:

- Header 18041602 (twitter.com.):
 - The timestamp hour is 18. This falls into the afternoon range (12:00-19:59).
 - The rule for "afternoon" sets `ip_pool_start` = 5 and `hash_mod` = 5.
 - The query ID is 02.
 - The hash index is calculated as $02 \% 5 = 2$.
 - The final index in the IP Pool is `ip_pool_start` + `hash_index` = 5 + 2 = 7.
 - The IP address at index 7 is 192.168.1.8. This matches the result in the table.
- Header 03324400 (_apple-mobdev._tcp.local.):
 - The timestamp hour is 03. This falls into the night range (20:00-03:59).

- The rule for "night" sets `ip_pool_start = 10` and `hash_mod = 5`.
- The query ID is 00.
- The hash index is calculated as $00 \% 5 = 0$.
- The final index in the IP Pool is $\text{ip_pool_start} + \text{hash_index} = 10 + 0 = 10$.
- The IP address at index 10 is 192.168.1.11. This matches the result in the table.

1.6. CONCLUSION

This task was completed successfully. The final implementation is a robust and correct DNS resolution system that demonstrates proficiency in network programming, protocol design, and low-level packet manipulation. The most critical lesson was the necessity of application-level mechanisms, such as echoing headers, to ensure correct request-response correlation over connectionless protocols like UDP. The system correctly parses, forwards, and resolves DNS queries according to all specified rules.

1.7. REPOSITORY

The complete source code and project files are available in the public GitHub repository: <https://github.com/rayvego/cs331-assignment-1>

1.8. REFERENCES

- Postel, J. (1981). RFC 791: Internet Protocol. Internet Engineering Task Force.
- Postel, J. (1980). RFC 768: User Datagram Protocol. Internet Engineering Task Force.
- Mockapetris, P. (1987). RFC 1035: Domain Names - Implementation and Specification. Internet Engineering Task Force.
- Braden, R. (1988). RFC 1071: Computing the Internet Checksum. Internet Engineering Task Force.

2. TASK 2: TRACEROUTE PROTOCOL BEHAVIOUR

2.1. INTRODUCTION AND METHODOLOGY

The purpose of this task is to understand and analyze the underlying network behavior of the traceroute utility across different operating systems. Experiments were conducted using a Linux virtual machine (Ubuntu 24.04) and a Windows 11 virtual machine. The network traffic generated by each system's respective traceroute command (traceroute on Linux, tracert on Windows) was captured using Wireshark. By filtering and

inspecting these captures, the specific protocols and packet fields used by each implementation can be identified and compared.

2.2. PROTOCOL ANALYSIS

The first objective is to answer the question: *What protocol does Windows tracert use by default, and what protocol does Linux traceroute use by default?*

Analysis of the Wireshark captures provides a definitive answer. The Linux traceroute command initiates its probes by sending a series of UDP packets to the destination. A key characteristic is the use of high-numbered destination ports, typically starting in the range of 33434. Conversely, the Windows tracert utility operates using the ICMP protocol. The capture shows it sends a series of "Echo (ping) request" packets to the destination. The difference in protocol choice is a fundamental design distinction between the two operating systems' implementations.

No.	Time	Source	Destination	Protocol	Length	Info
7	3.5413125...	10.7.22.178	104.16.123.96	UDP	74	59478 → 33434 Len=32
8	3.5413391...	10.7.22.178	104.16.123.96	UDP	74	37276 → 33435 Len=32
9	3.5413480...	10.7.22.178	104.16.123.96	UDP	74	46100 → 33436 Len=32
10	3.5413592...	10.7.22.178	104.16.123.96	UDP	74	59905 → 33437 Len=32
11	3.5413684...	10.7.22.178	104.16.123.96	UDP	74	57081 → 33438 Len=32
12	3.5413793...	10.7.22.178	104.16.123.96	UDP	74	38562 → 33439 Len=32
13	3.5413926...	10.7.22.178	104.16.123.96	UDP	74	41438 → 33440 Len=32
14	3.5414027...	10.7.22.178	104.16.123.96	UDP	74	45675 → 33441 Len=32
15	3.5414159...	10.7.22.178	104.16.123.96	UDP	74	55534 → 33442 Len=32
16	3.5414261...	10.7.22.178	104.16.123.96	UDP	74	54870 → 33443 Len=32
17	3.5414375...	10.7.22.178	104.16.123.96	UDP	74	59516 → 33444 Len=32
18	3.5414447...	10.7.22.178	104.16.123.96	UDP	74	52485 → 33445 Len=32
19	3.5414571...	10.7.22.178	104.16.123.96	UDP	74	50212 → 33446 Len=32
20	3.5414686...	10.7.22.178	104.16.123.96	UDP	74	60600 → 33447 Len=32
21	3.5414777...	10.7.22.178	104.16.123.96	UDP	74	39575 → 33448 Len=32
22	3.5414902...	10.7.22.178	104.16.123.96	UDP	74	56569 → 33449 Len=32
23	3.5491807...	172.16.4.7	10.7.22.178	ICMP	102	Time-to-live exceeded (Time to live exceeded in transit)
24	3.5491809...	172.16.4.7	10.7.22.178	ICMP	102	Time-to-live exceeded (Time to live exceeded in transit)
25	3.5491810...	172.16.4.7	10.7.22.178	ICMP	102	Time-to-live exceeded (Time to live exceeded in transit)
26	3.5491810...	10.7.0.5	10.7.22.178	ICMP	70	Time-to-live exceeded (Time to live exceeded in transit)
27	3.5491810...	10.117.81.253	10.7.22.178	ICMP	70	Time-to-live exceeded (Time to live exceeded in transit)
28	3.5491810...	10.7.0.5	10.7.22.178	ICMP	70	Time-to-live exceeded (Time to live exceeded in transit)
29	3.5491810...	10.117.81.253	10.7.22.178	ICMP	70	Time-to-live exceeded (Time to live exceeded in transit)
30	3.5491810...	10.117.81.253	10.7.22.178	ICMP	70	Time-to-live exceeded (Time to live exceeded in transit)
31	3.5492212...	10.7.0.5	10.7.22.178	ICMP	70	Time-to-live exceeded (Time to live exceeded in transit)
32	3.5492212...	14.139.98.1	10.7.22.178	ICMP	70	Time-to-live exceeded (Time to live exceeded in transit)
33	3.5492212...	14.139.98.1	10.7.22.178	ICMP	70	Time-to-live exceeded (Time to live exceeded in transit)
34	3.5499230...	14.139.98.1	10.7.22.178	ICMP	70	Time-to-live exceeded (Time to live exceeded in transit)
37	3.5564788...	10.7.22.178	104.16.123.96	UDP	74	33123 → 33450 Len=32
38	3.5565004...	10.7.22.178	104.16.123.96	UDP	74	41732 → 33451 Len=32
39	3.5565112...	10.7.22.178	104.16.123.96	UDP	74	58338 → 33452 Len=32
40	3.5565382...	10.7.22.178	104.16.123.96	UDP	74	40250 → 33453 Len=32
41	3.5565499...	10.7.22.178	104.16.123.96	UDP	74	39411 → 33454 Len=32
42	3.5565580...	10.7.22.178	104.16.123.96	UDP	74	48283 → 33455 Len=32
43	3.5565689...	10.7.22.178	104.16.123.96	UDP	74	38016 → 33456 Len=32
44	3.5565750...	10.7.22.178	104.16.123.96	UDP	74	41377 → 33457 Len=32
51	3.5779992...	10.119.234.162	10.7.22.178	ICMP	110	Time-to-live exceeded (Time to live exceeded in transit)
52	3.5779992...	10.119.234.162	10.7.22.178	ICMP	110	Time-to-live exceeded (Time to live exceeded in transit)
53	3.5779992...	10.119.234.162	10.7.22.178	ICMP	110	Time-to-live exceeded (Time to live exceeded in transit)
54	3.5785681...	10.7.22.178	104.16.123.96	UDP	74	38763 → 33458 Len=32
55	3.5785807...	10.7.22.178	104.16.123.96	UDP	74	55817 → 33459 Len=32
56	3.5785939...	10.7.22.178	104.16.123.96	UDP	74	34388 → 33460 Len=32
57	3.5786028...	10.7.22.178	104.16.123.96	UDP	74	59025 → 33461 Len=32
58	3.5786280...	10.7.22.178	104.16.123.96	UDP	74	41122 → 33462 Len=32
59	3.5786369...	10.7.22.178	104.16.123.96	UDP	74	57817 → 33463 Len=32
60	3.5786486...	10.7.22.178	104.16.123.96	UDP	74	43735 → 33464 Len=32
61	3.6845521...	104.23.231.7	10.7.22.178	ICMP	70	Time-to-live exceeded (Time to live exceeded in transit)
62	3.6845524...	104.23.231.30	10.7.22.178	ICMP	70	Time-to-live exceeded (Time to live exceeded in transit)
63	3.6845524...	104.23.231.30	10.7.22.178	ICMP	70	Time-to-live exceeded (Time to live exceeded in transit)
64	3.6845524...	103.218.244.94	10.7.22.178	ICMP	70	Time-to-live exceeded (Time to live exceeded in transit)
65	3.6845525...	103.218.244.94	10.7.22.178	ICMP	70	Time-to-live exceeded (Time to live exceeded in transit)
66	3.6845525...	103.218.244.94	10.7.22.178	ICMP	70	Time-to-live exceeded (Time to live exceeded in transit)
67	3.6845525...	104.16.123.96	10.7.22.178	ICMP	102	Destination unreachable (Port unreachable)
68	3.6845557...	10.7.22.178	104.16.123.96	UDP	74	60190 → 33465 Len=32
69	3.6846744...	10.7.22.178	104.16.123.96	UDP	74	57567 → 33466 Len=32
70	3.7225666...	104.16.123.96	10.7.22.178	ICMP	102	Destination unreachable (Port unreachable)
71	3.7225667...	104.16.123.96	10.7.22.178	ICMP	102	Destination unreachable (Port unreachable)

Figure 4: Wireshark capture of Linux traceroute showing default UDP probes.

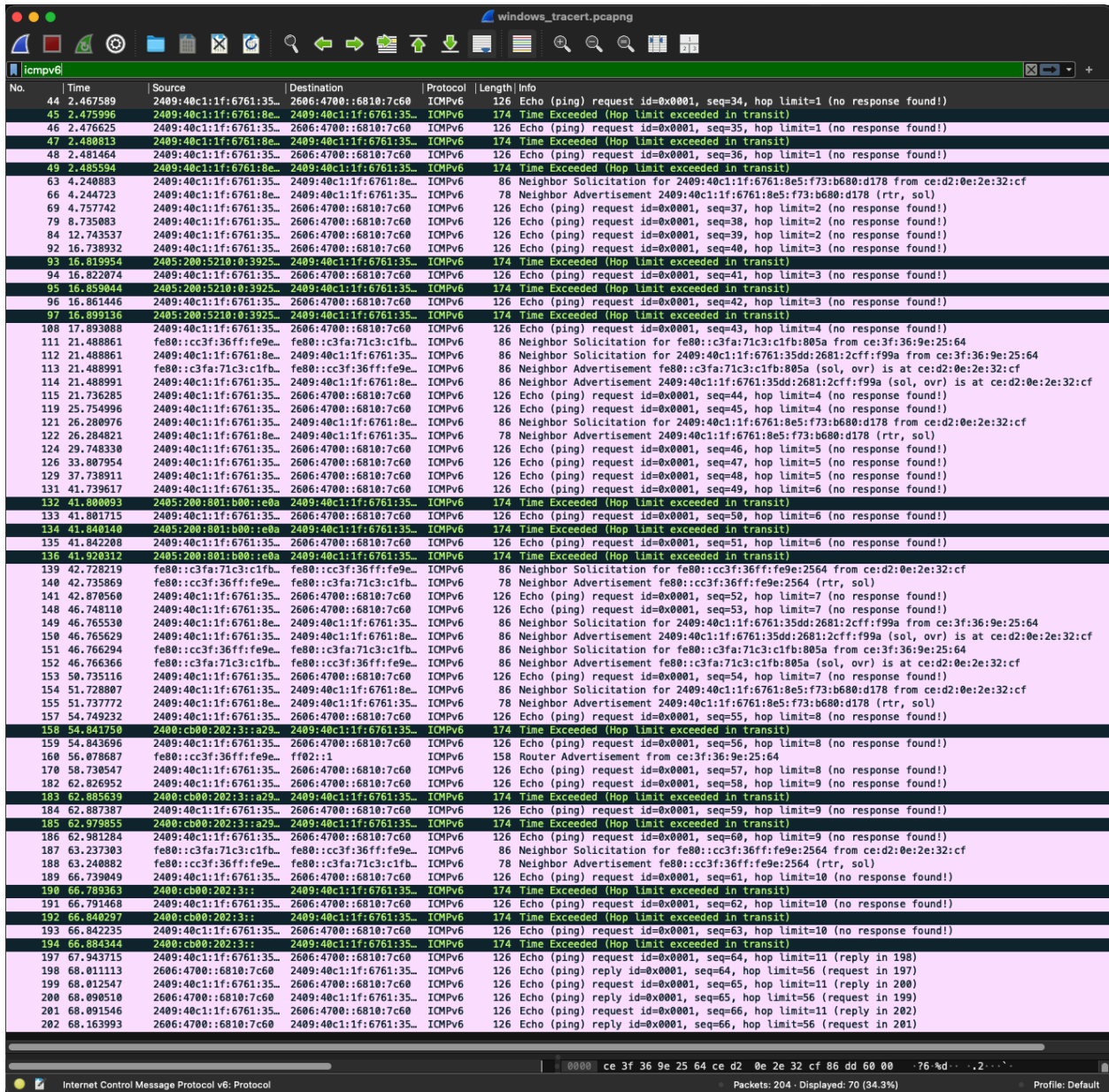


Figure 5: Wireshark capture of Windows tracert showing default ICMPv6 probes.

2.3. ANALYSIS OF NON-RESPONSIVE HOPS

The traceroute outputs for both operating systems contained hops that appeared as * * *. This indicates that a router at that hop in the path did not send a reply back to our machine within the expected time frame. There are two primary technical reasons for this behavior:

1. Firewall Filtering: It is a common security practice for network administrators to

configure routers or firewalls to drop the types of packets used by traceroute. This is done to prevent outsiders from mapping the internal structure of their network.

2. **ICMP Rate Limiting:** Routers are designed to forward user traffic as their top priority. Responding to diagnostic requests like traceroute is a low-priority task. To protect themselves from being overwhelmed, routers often limit the number of ICMP error messages (like "Time-to-live exceeded") they will send in a given period. If a router is busy or receives many probes, it may simply ignore them, resulting in a timeout on the user's end.

```
~ (45.598s)
traceroute www.cloudflare.com
traceroute: Warning: www.cloudflare.com has multiple addresses; using 104.16.124.96
traceroute to www.cloudflare.com (104.16.124.96), 64 hops max, 40 byte packets
 1  10.7.0.5 (10.7.0.5)  3.516 ms  4.256 ms  3.048 ms
 2  172.16.4.7 (172.16.4.7)  4.425 ms  3.543 ms  11.741 ms
 3  14.139.98.1 (14.139.98.1)  5.507 ms  5.613 ms  5.384 ms
 4  10.117.81.253 (10.117.81.253)  3.192 ms  3.030 ms  4.312 ms
 5  * * *
 6  * * *
 7  * * *
 8  10.119.234.162 (10.119.234.162)  21.550 ms  22.168 ms  23.305 ms
 9  103.218.244.94 (103.218.244.94)  42.300 ms  34.515 ms  44.800 ms
10  104.23.231.5 (104.23.231.5)  33.567 ms
    104.23.231.30 (104.23.231.30)  32.185 ms
    104.23.231.7 (104.23.231.7)  32.125 ms
11  104.16.124.96 (104.16.124.96)  31.802 ms  31.623 ms  31.551 ms
```

Figure 6: Traceroute output showing non-responsive hops (hops 5, 6, and 7).

2.3.1. WHY NON-RESPONSIVE HOPS DO NOT TERMINATE THE TRACE

The appearance of *** in the traceroute output indicates that the router did not return an ICMP response to the probe packet. This does not imply that the router dropped or blocked the packet in a manner that prevents forwarding to subsequent hops.

Routers prioritize packet forwarding as their primary function. Upon receiving a packet with a TTL greater than 1, the router decrements the TTL and forwards the packet, irrespective of whether it generates an ICMP error message. The generation of an ICMP "Time-to-live exceeded" response is an optional behavior, as specified in IP standards such as RFC 791. Many routers are configured to suppress these responses for security purposes (to obscure network topology) or due to ICMP rate limiting.

For a probe packet with TTL = n (targeting hop n):

- If the router at hop n does not reply with an ICMP message, the output displays

***.

- However, for the subsequent probe with $TTL = n+1$, the same router will decrement the TTL and forward the packet to hop $n+1$, provided the TTL remains greater than 1 upon arrival.

In essence, the router continues to route and forward packets normally but disregards the request for an ICMP reply. This permits the traceroute process to proceed and identify later hops.

If a router were to block or drop all traffic (e.g., due to a firewall policy), the trace would fail from that point onward, resulting in *** for all remaining hops and an inability to reach the destination.

2.4. ANALYSIS OF PROBE PACKET FIELDS

The next question is: *In Linux traceroute, which field in the probe packets changes between successive probes sent to the destination?*

The core mechanism of traceroute relies on the manipulation of the Time To Live (TTL) field within the IP header of the outgoing probe packets. For the first hop, traceroute sends a packet with a TTL value of 1. For the second hop, it sends a new packet with a TTL of 2, and so on. When a router receives a packet, it decrements the TTL by one. If the TTL becomes zero, the router discards the packet and sends an ICMP "Time-to-live exceeded" error back to the source. By analyzing which router sends this error, traceroute can map the path step-by-step. The Wireshark capture confirms this behavior, showing the TTL value incrementing for each subsequent hop.

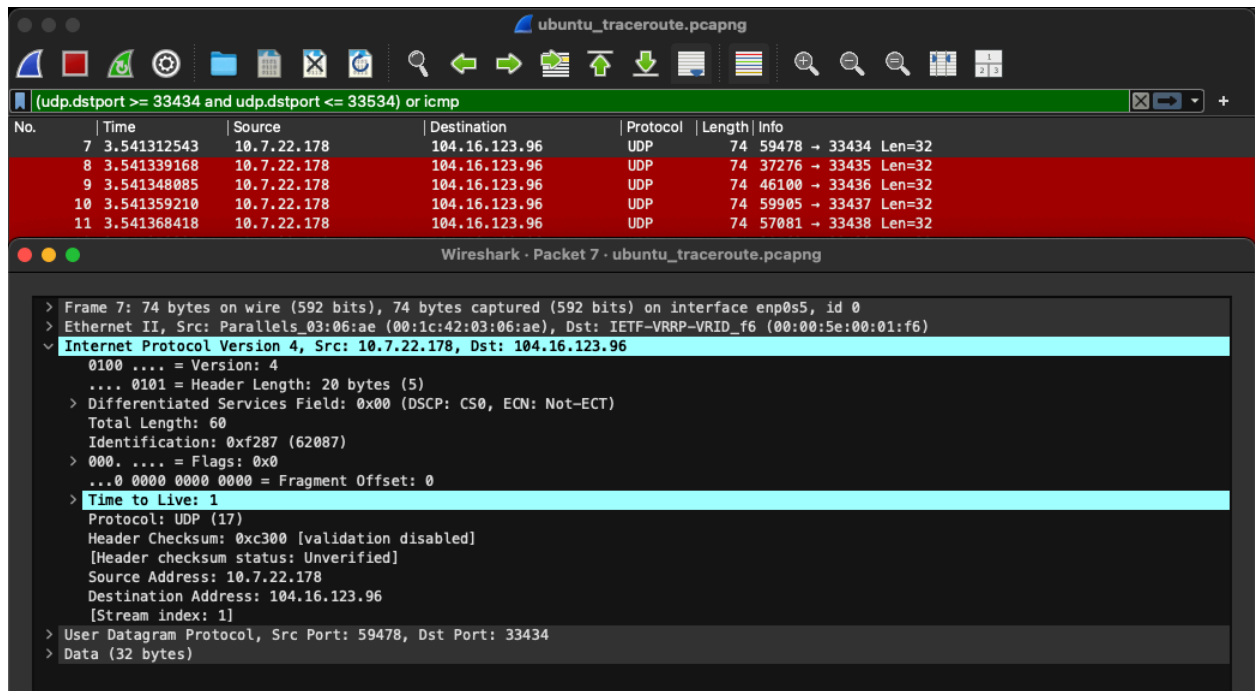


Figure 7: First Linux probe packet showing TTL value of 1.

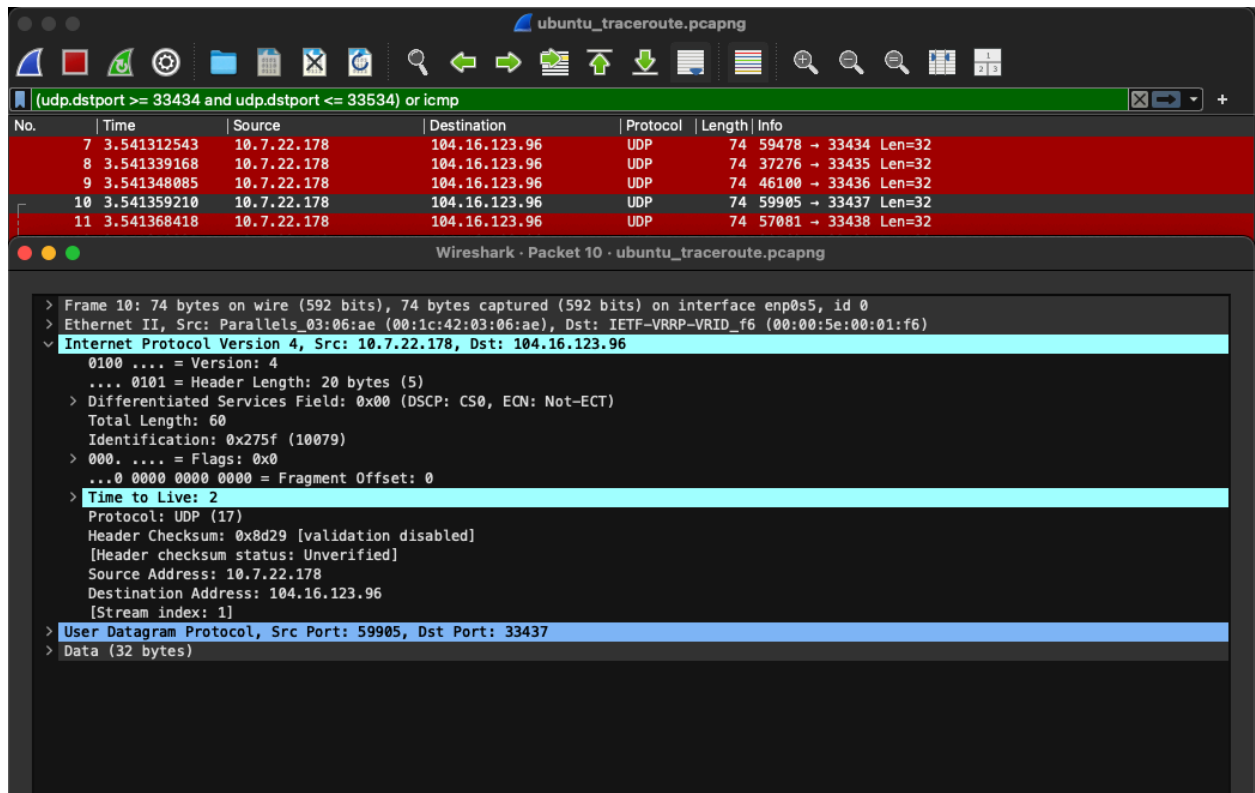


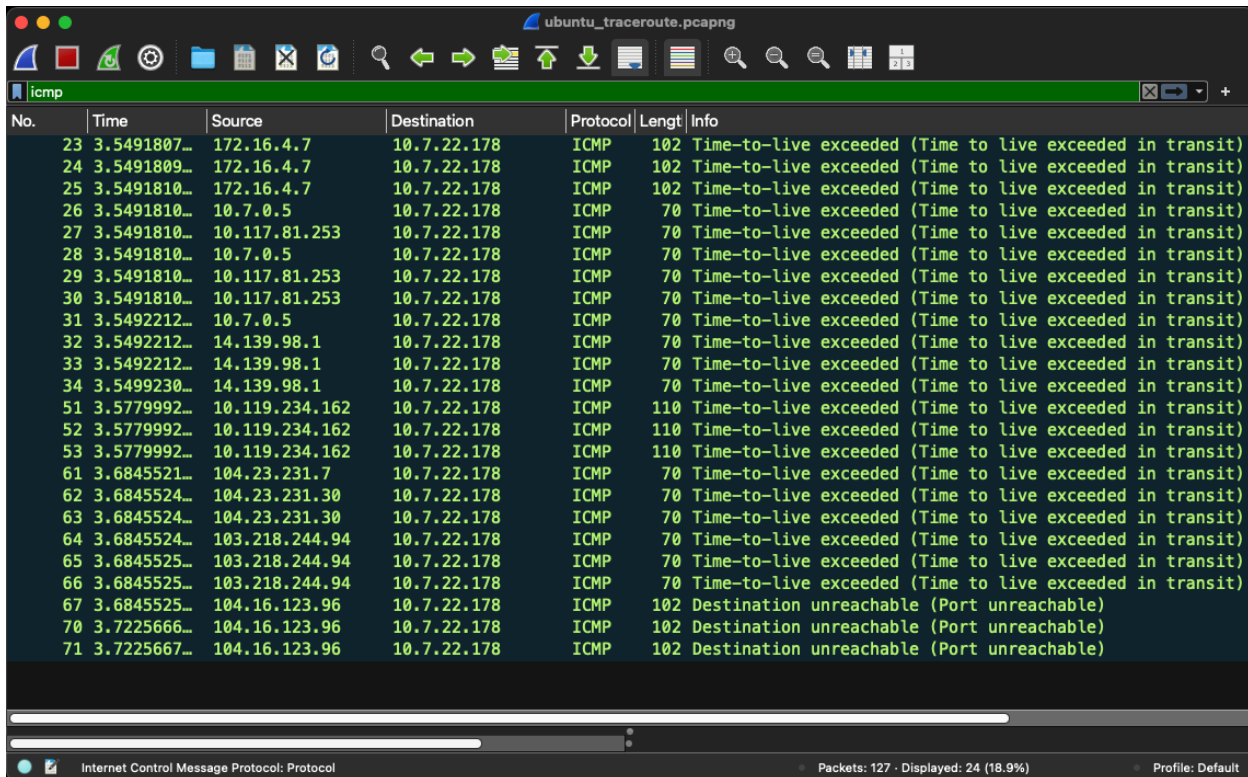
Figure 8: Subsequent Linux probe packet showing TTL value of 2.

2.5. ANALYSIS OF FINAL HOP RESPONSE

The final question is: *At the final hop, how is the response different compared to the intermediate hop?*

There is a distinct difference between the ICMP message received from an intermediate router and the message received from the final destination server in a Linux/macOS UDP-based traceroute.

- Intermediate Hops: As described previously, routers along the path whose TTL decrement causes the packet to expire respond with an ICMP message of "Time to live exceeded" (Type 11).
- Final Hop: When a UDP probe finally reaches the intended destination, the server's operating system receives it on a high-numbered, unused port. Since no application is listening there, the OS kernel responds with an ICMP error message of "Destination unreachable (Port unreachable)" (Type 3, Code 3). This "error" is the specific signal that the traceroute program uses to determine that it has successfully reached the final destination and can terminate the trace.



No.	Time	Source	Destination	Protocol	Length	Info
23	3.5491807...	172.16.4.7	10.7.22.178	ICMP	102	Time-to-live exceeded (Time to live exceeded in transit)
24	3.5491809...	172.16.4.7	10.7.22.178	ICMP	102	Time-to-live exceeded (Time to live exceeded in transit)
25	3.5491810...	172.16.4.7	10.7.22.178	ICMP	102	Time-to-live exceeded (Time to live exceeded in transit)
26	3.5491810...	10.7.0.5	10.7.22.178	ICMP	70	Time-to-live exceeded (Time to live exceeded in transit)
27	3.5491810...	10.117.81.253	10.7.22.178	ICMP	70	Time-to-live exceeded (Time to live exceeded in transit)
28	3.5491810...	10.7.0.5	10.7.22.178	ICMP	70	Time-to-live exceeded (Time to live exceeded in transit)
29	3.5491810...	10.117.81.253	10.7.22.178	ICMP	70	Time-to-live exceeded (Time to live exceeded in transit)
30	3.5491810...	10.117.81.253	10.7.22.178	ICMP	70	Time-to-live exceeded (Time to live exceeded in transit)
31	3.5492212...	10.7.0.5	10.7.22.178	ICMP	70	Time-to-live exceeded (Time to live exceeded in transit)
32	3.5492212...	14.139.98.1	10.7.22.178	ICMP	70	Time-to-live exceeded (Time to live exceeded in transit)
33	3.5492212...	14.139.98.1	10.7.22.178	ICMP	70	Time-to-live exceeded (Time to live exceeded in transit)
34	3.5499230...	14.139.98.1	10.7.22.178	ICMP	70	Time-to-live exceeded (Time to live exceeded in transit)
51	3.5779992...	10.119.234.162	10.7.22.178	ICMP	110	Time-to-live exceeded (Time to live exceeded in transit)
52	3.5779992...	10.119.234.162	10.7.22.178	ICMP	110	Time-to-live exceeded (Time to live exceeded in transit)
53	3.5779992...	10.119.234.162	10.7.22.178	ICMP	110	Time-to-live exceeded (Time to live exceeded in transit)
61	3.6845521...	104.23.231.7	10.7.22.178	ICMP	70	Time-to-live exceeded (Time to live exceeded in transit)
62	3.6845524...	104.23.231.30	10.7.22.178	ICMP	70	Time-to-live exceeded (Time to live exceeded in transit)
63	3.6845524...	104.23.231.30	10.7.22.178	ICMP	70	Time-to-live exceeded (Time to live exceeded in transit)
64	3.6845524...	103.218.244.94	10.7.22.178	ICMP	70	Time-to-live exceeded (Time to live exceeded in transit)
65	3.6845525...	103.218.244.94	10.7.22.178	ICMP	70	Time-to-live exceeded (Time to live exceeded in transit)
66	3.6845525...	103.218.244.94	10.7.22.178	ICMP	70	Time-to-live exceeded (Time to live exceeded in transit)
67	3.6845525...	104.16.123.96	10.7.22.178	ICMP	102	Destination unreachable (Port unreachable)
70	3.7225666...	104.16.123.96	10.7.22.178	ICMP	102	Destination unreachable (Port unreachable)
71	3.7225667...	104.16.123.96	10.7.22.178	ICMP	102	Destination unreachable (Port unreachable)

Figure 9: Comparison of ICMP responses

2.6. HYPOTHETICAL FIREWALL SCENARIO

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

In this scenario, the outcomes for the two utilities would be completely different:

- Windows tracert: This tool would work without any issues. Its functionality is based entirely on ICMP packets for both its outgoing probes ("Echo request") and the incoming responses ("Time-to-live exceeded" and "Echo reply"). As the firewall permits all ICMP traffic, the trace would complete successfully.
- Linux traceroute: In its default UDP mode, this tool would fail completely. The firewall would block every outgoing UDP probe packet at the network edge. The program would send its probes but never receive any "Time-to-live exceeded" responses, because the probes never reach the routers. The output would consist entirely of * * * for every hop until the program times out. However, it should be noted that the Linux utility can be instructed to use ICMP probes with the -I flag, in which case it would also succeed.

2.7. REFERENCES

- <https://linux.die.net/man/8/traceroute>
- <https://wiki.wireshark.org/Traceroute>
- <https://tools.ietf.org/html/rfc792>
- <https://tools.ietf.org/html/rfc791>