Indian Institute of Technology Gandhinagar

Department of Computer Science and Engineering

CS-331: Computer Networks

Assignment 1

Submitted by:
Anurag Singh (23110035)
Arjun B Dikshit (23110040)

Instructor:
Prof. Sameer Kulkarni

September 15, 2025

Contents

1 Introduction

The purpose of this report is to explore and implement fundamental networking concepts through two practical tasks. In the first task, we designed and deployed a custom DNS resolution system consisting of a client and a server. The system was required to process DNS queries from a packet capture file, augment them with a custom header, and resolve them into predefined IP addresses using server-side rules. This exercise provided practical experience with packet parsing, socket communication, and DNS protocol manipulation.

The second task focused on analyzing the behavior of the `tracert` and `traceroute` utilities across two different operating systems: Windows and Linux. By capturing and studying the packets exchanged during route tracing, we investigated the default protocols employed by each tool, the causes of packet loss or missing hops, and the differences in response behavior between intermediate and final nodes. These experiments deepened our understanding of how various tools like Wireshark use different types of filters and rely on core Internet protocols such as ICMP and UDP to discover network paths.

Together, the two questions combined implementation and analysis, allowing us to get a deeper understanding of sockets and packet transfer protocols.

2 Question 1

2.1 Methodology

In this question, we were required to design and implement a custom DNS resolution system comprising a client and a server. We build a client that processes an input pcap file to get DNS query packets, appends a custom header, and sends the new packet to the server which resolves them to specific IP addresses according to a predefined set of rules.

2.1.1 Client Implementation

The client application was developed to perform the following sequential steps:

1. **PCAP File Parsing and Filtering:** The client first reads the "5.pcap" file using the `dpkt` library. It then filters the raw packets to isolate only valid DNS query packets based on a strict set of criteria, including verifying that the packet is a UDP packet with a source or destination port of 53 and that the DNS query flag (`qr`) is set to 0.
2. **Custom Header Addition:** A unique 8-byte custom header was generated for each filtered DNS query packet. This header is formatted as `HHMMSSID`, where `HH` is the current hour in 24-hour format, `MM` is the minute, `SS` is the second, and `ID` is a two-digit sequence ID starting from 00. This custom header was then prepended to the original DNS packet.
3. **Communication with Server:** The client established a TCP connection with the server at `127.0.0.1` on port `23354`. The chosen port was made sure to be between 1024 – 49151, so that it lies in usable ports range.
4. **Response Logging:** Upon receiving a response, the client parsed the packet to extract the resolved domain and IP address. The custom header, domain, and resolved IP were logged and stored to be included in the final report.

2.1.2 Server Implementation

The server was designed to handle incoming requests as follows:

1. **Socket Initialization:** The server initialized a TCP socket and started listening for at most connections on port `23354`.
2. **Header Extraction and Rule Application:** Upon receiving a packet, the server decoded the payload to extract the custom header. It used the header's timestamp and packet ID to apply a set of predefined rules from a `rules.json` file. These rules determined which IP address from a given pool should be used for resolution.
3. **DNS Response Generation:** The server then constructed a DNS response packet. This involved modifying the original query packet by setting the response flag (`qr`) and adding an answer section containing the resolved IP address determined by the rules.
4. **Packet Transmission:** The newly generated DNS response packet was then sent back to the client.

2.2 Results

The client successfully processed **6 DNS query packets** from the PCAP file. The server correctly resolved each query to a designated IP address. The results of the DNS resolution are presented in the table below.

Table 2.1: DNS Queries and Resolved IP Addresses

Custom Header	Domain	Resolved IP
15290300	apple.com	192.168.1.6
15290301	facebook.com	192.168.1.7
15290302	amazon.com	192.168.1.8
15290303	twitter.com	192.168.1.9
15290304	wikipedia.org	192.168.1.10
15290305	stackoverflow.com	192.168.1.6

The following figure shows the DNS query packets sent by the client and the final DNS packet received by the client containing resolved IP in answers section of DNS packet.

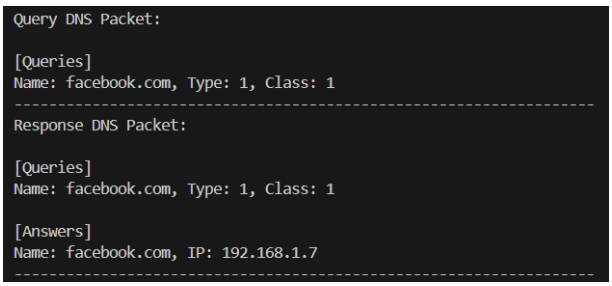

Figure 2.1: DNS packets before and after

The system demonstrated a successful end-to-end flow, from packet capture and filtering to custom header addition, communication, and rule-based DNS resolution. The total number of DNS query packets processed was 6.

3 Question 2

3.1 Methodology

To analyze the behavior of the `tracert` and `traceroute` utilities on different operating systems, experiments were conducted on both Windows and Linux environments. The objective was to understand the underlying protocols used, the reasons for potential packet loss, and the differences in responses at intermediate and final hops.

The following steps were performed for both operating systems, targeting the destination `www.discord.com` (with an IP address of `162.159.135.232`):

1. **Command Execution:** The `tracert www.discord.com` command was executed on a Windows machine. Subsequently, the `traceroute www.discord.com` command was executed on a Linux machine. Both commands were configured to trace the route with a maximum of 30 hops.
2. **Network Traffic Capture:** Wireshark was used to capture all network traffic during the execution of both commands. The captures were then analyzed to inspect the protocols, packet fields, and response types. The captures were saved and labeled for later analysis.
3. **Data Analysis:** The captured packets were examined in detail. Key fields such as the Protocol field, Time-to-Live (TTL) value, and the type of ICMP messages received were specifically analyzed to answer the posed questions.

3.2 Observations

3.2.1 Windows

We run the **tracert** routine on a Windows machine with the destination being **discord**. We get the following output:

```
C:\Users\Arjun>tracert www.discord.com

Tracing route to www.discord.com [162.159.135.232]
over a maximum of 30 hops:

  1     3 ms     2 ms     2 ms  10.240.0.2
  2    16 ms     2 ms     3 ms  10.3.0.29
  3     9 ms     5 ms     5 ms  10.3.0.5
  4    12 ms     2 ms     2 ms  172.16.4.7
  5    15 ms     4 ms     5 ms  14.139.98.1
  6    13 ms     2 ms     4 ms  10.117.81.253
  7     *        *        *     Request timed out.
  8     *        *        *     Request timed out.
  9     *        *        *     Request timed out.
 10    22 ms    23 ms    22 ms  10.119.234.162
 11    45 ms    35 ms    35 ms  103.218.244.94
 12    41 ms    36 ms    35 ms  104.23.231.30
 13    42 ms    36 ms    35 ms  162.159.135.232

Trace complete.
```

Figure 3.1: Output of running tracert on windows

We capture the packets using Wireshark (and filter for **ICMP**)

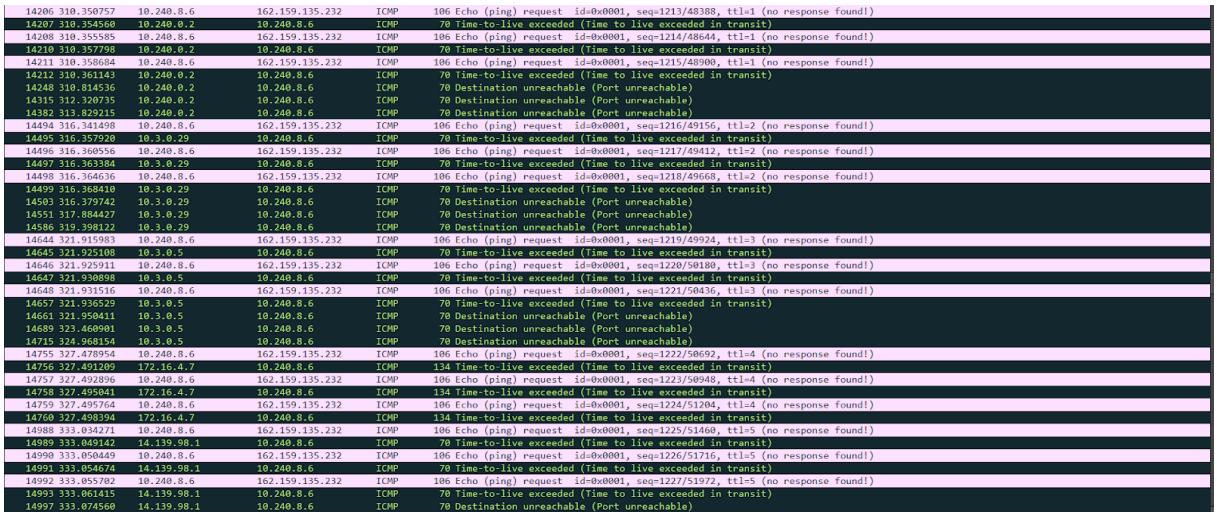

Figure 3.2: Wireshark Captures with packets (TTL 1 to TTL 5)

In Figure 3.2, we can see that every failure evokes 2 responses from the router where TTL expired

1. TTL exceeded in transit
2. Destination Unreachable

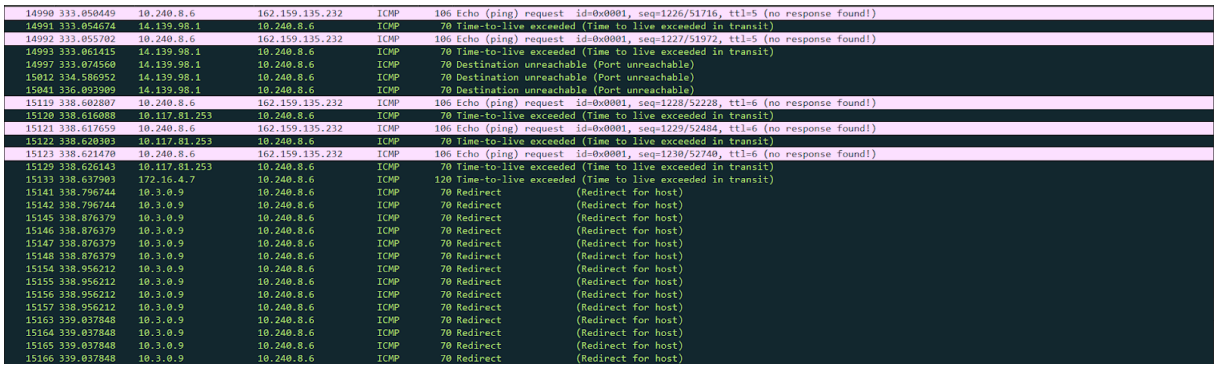

Figure 3.3: Wireshark Captures with packets (TTL 5 to TTL 6)

In Figure 3.3, we see some Redirect from Host messages. This is not a property of the routine, but a property of the ISP, (which in this case happens to be IITGN SSO). This is a common optimization on campus/enterprise networks. For the sake of our analysis, we ignore these.

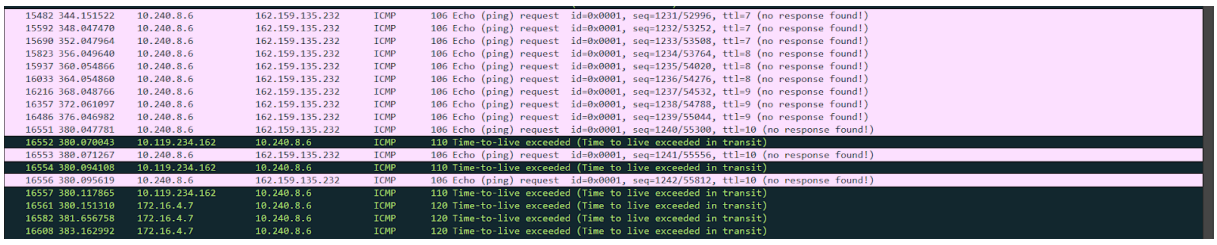

Figure 3.4: Wireshark Captures with packets (TTL 7 to TTL 10)

In Figure 3.5, we can finally see here that, on the 13th hop, we reached the destination server. The destination server sends back an **ICMP Echo Reply** to the request we have issued.

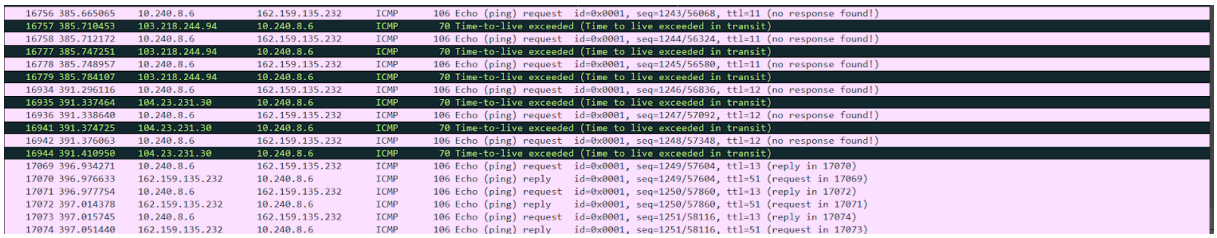

Figure 3.5: Wireshark Captures with packets (TTL 11 to TTL 13)

3.2.2 Linux

On Linux, we run the **traceroute** tool with **discord** as the destination. In the output, each line corresponds to a hop along the network path, listing the router's IP address and the round-trip times (RTT) for three probes. Some hops return * * *, meaning they did not respond, likely due to ICMP being blocked. The trace successfully reaches the Discord server at hop 13 (162.159.135.232), showing the complete path from the local machine to the destination.

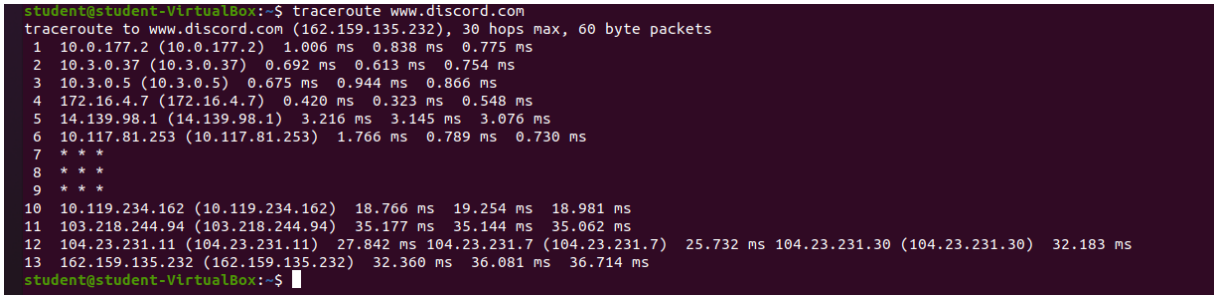

Figure 3.6: Output of traceroute

In figure 3.7, we can see the **Wireshark** capture of packets, generated by the Linux traceroute. The destination ports (33459, 33460, 33461, etc.) are deliberately varied for each probe. This variation allows traceroute to uniquely identify the corresponding ICMP replies for each packet.

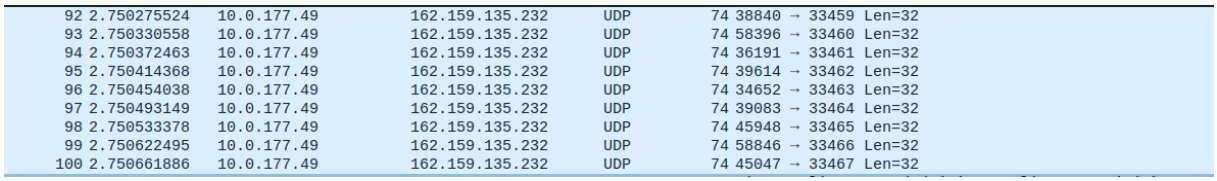

Figure 3.7: Port increments for each probe

The figure 3.8 shows that traceroute sends UDP packets from the source (10.0.177.49) to the destination (162.159.135.232). Intermediate routers respond with ICMP "Time-to-live exceeded" messages when the packet's TTL reaches zero, which traceroute uses to discover each hop. Finally, the destination host replies with ICMP "Port unreachable", confirming that the probes reached the target since no application is listening on the specified ports.

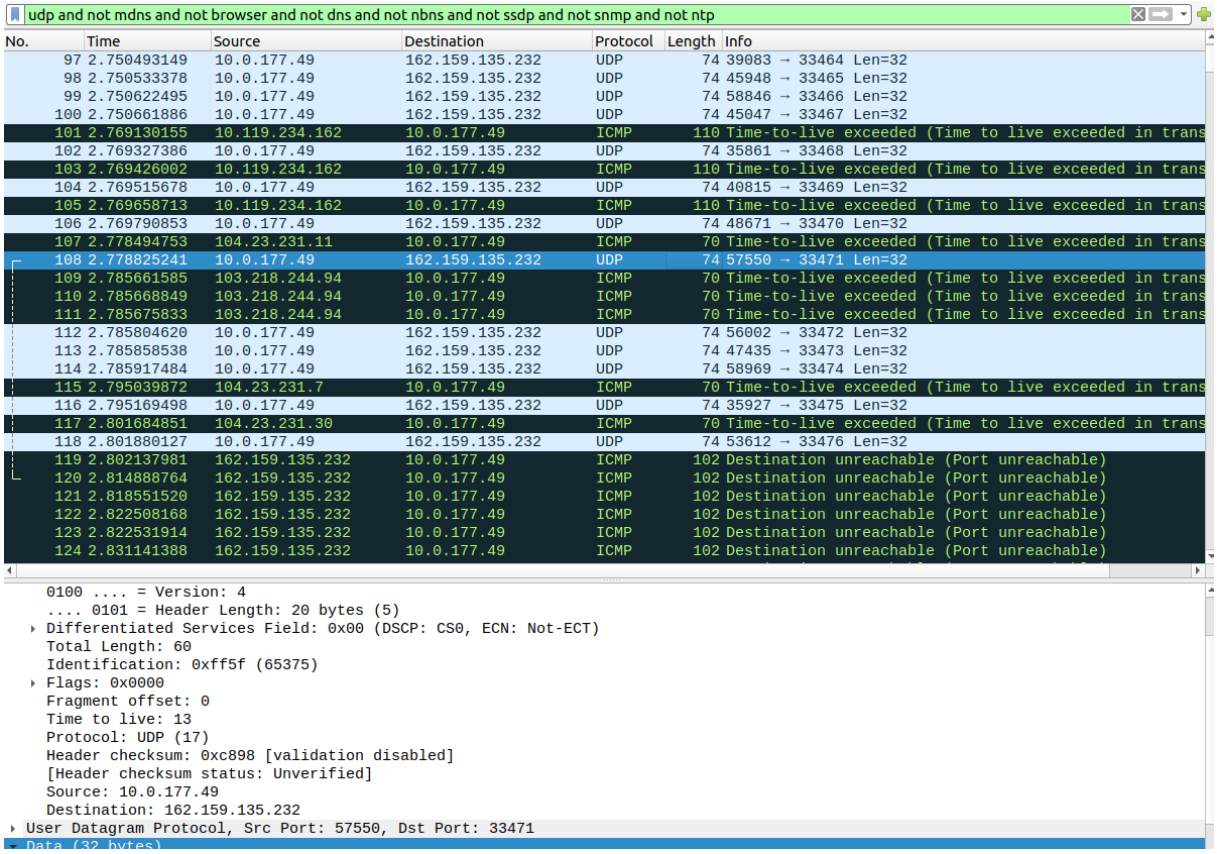

Figure 3.8: Wireshark captures UDP requests & ICMP replies

3.3 Results

Based on the observations from the network traffic captures, the following answers were formulated:

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

   - **Windows `tracert`** uses the **Internet Control Message Protocol (ICMP)** by default. It sends a series of ICMP Echo Request packets to trace the route.
   - **Linux `traceroute`** uses the **User Datagram Protocol (UDP)** by default. It sends UDP probe packets to a high, unlikely destination port number.

2. **Some hops in your traceroute output may show ***. Provide at least two reasons why a router might not reply**
   The ***, indicating a "request timed out," occurs when a router along the path does not send a reply packet within the expected time frame. Based on the captures, two primary reasons for this are:

   - **Firewall or ACL (Access Control List):** The router's firewall or ACL might be configured to drop ICMP or UDP packets to prevent denial-of-service attacks or to hide network topology. This would prevent the router from sending the necessary ICMP Time-to-live exceeded message back to the source.
   - **Router Prioritization:** The router may be configured to prioritize routing functions

over responding to ICMP or UDP probe packets. During periods of high traffic or heavy load, the router might drop these probe packets to focus on critical data forwarding tasks.

3. **In Linux traceroute, which field in the probe packets changes between successive probes sent to the destination?**
   In Linux `traceroute`, the **destination port number** of the UDP probe packets changes between successive probes. The initial probes for a specific hop start at a high port number (e.g., 33434) and increment with each subsequent probe. This mechanism, combined with the incrementing TTL, allows the utility to identify intermediate routers.

4. **At the final hop, how is the response different compared to the intermediate hop?**
   The response received from the final hop is fundamentally different from the responses from intermediate hops.

   - **Intermediate Hops:** Intermediate routers respond with an **ICMP Time-to-live exceeded in transit** message. This is because the probe packet's TTL value reaches zero at that hop, and the router discards the packet and sends this error message back to the source.
   - **Final Hop:** The final destination host responds with a different message. For Linux `traceroute` (using UDP), since the destination port is an unlikely, unlistened-to port, the host responds with an **ICMP Destination Unreachable (Port Unreachable)** message. For Windows `tracert` (using ICMP Echo), the final hop responds with an **ICMP Echo Reply**. This indicates that the probe packet successfully reached its intended destination.

5. **Suppose a firewall blocks UDP traffic but allows ICMP — how would this affect the results of Linux traceroute vs. Windows tracert?**
   Assuming a firewall is present on the destination router only, if a firewall blocks UDP traffic but allows ICMP:

   - **Windows `tracert`** would be **unaffected**. Since it relies on ICMP Echo Request packets and ICMP Time-to-live exceeded messages, its functionality would remain intact.
   - **Linux `traceroute`** would be **affected**. The intermediate hops would still respond with ICMP Time-to-live exceeded messages, as this protocol is not blocked. However, the final destination host, which sends an ICMP Destination Unreachable message in response to a UDP packet, would not receive the probe packet, and thus no response would be returned. The traceroute would show `***`, timing out at the final hop.

4 Conclusion

This assignment provided valuable insights into both the design of network applications and the analysis of diagnostic tools.

The first task provided hands-on experience in building a functional client-server system for custom DNS resolution. We successfully implemented packet parsing logic to filter DNS queries from a PCAP file and designed a custom header for a timestamp-rule-based IP address resolution. The results demonstrate the working and correctness of the custom DNS resolver.

The second task, examining the behavior of `tracert` on Windows and `traceroute` on Linux, deepened our understanding of how different operating systems adopt distinct probing strategies. We understand the importance of protocol choice in this task as we examine the difference in routine behavior of tracert (using ICMP protocol) and traceroute (using the UDP protocol) Capturing and analyzing the traffic clarified why responses differ between intermediate hops and the destination, and how network policies such as UDP blocking influence results and how sometimes we might not get a response back from the relevant router.

Together, these tasks bridged the gap between theoretical networking concepts and their practical applications. They underscored the role of protocol details in shaping system behavior, highlighted the diagnostic power of tools like traceroute, and demonstrated the challenges posed by real-world network conditions. Overall, this work strengthened both conceptual understanding and applied proficiency in computer networking.