CS331: Computer Networks – Assignment 1

Custom DNS Resolver with Time-Based Load Balancing

Vedant Chichmalkar 22110282
vedant.chichmalkar.iitgn.ac.in

Rutvi Shah 22110227
rutvi.shah@iitgn.ac.in

September 14, 2025

Contents

1 Task 1

1.1 Introduction

The Domain Name System (DNS) is a core service of the internet, mapping human-readable domain names (e.g., `www.google.com`) to machine-readable IP addresses. This project implements a custom DNS resolver to explore packet-level analysis, protocol customization, and rule-based server logic.

The system follows a client-server architecture. The client parses a Packet Capture (`.pcap`) file to extract DNS queries, generates an 8-byte custom header (`HHMMSSID`), and forwards the modified packets via UDP. The server receives these packets, parses the header, and applies a time-based load-balancing algorithm. Instead of performing a standard DNS lookup, it selects an IP from a predefined pool based on rules specified in an external configuration file (`rules.json`).

The objective of this assignment is to demonstrate network programming concepts such as Scapy-based packet parsing, UDP socket communication, and custom application-layer protocol design to implement load balancing.

1.2 Methodology

The implementation is divided into three main components: the protocol design, the client's packet processing, and the server's resolution algorithm.

1.2.1 PCAP File Selection

As per the assignment guidelines, the PCAP file was chosen based on the sum of the last three digits of the team members' student IDs, modulo 10.

- Vedant Chichmalkar (22110**282**): 282
- Rutvi Shah (22110**227**): 227

$$\text{File Index} = (282 + 227)\%10 = 9$$

Therefore, the file `9.pcap` was used for this task.

1.2.2 System Architecture and Custom Protocol

Communication between client and server uses UDP, the standard transport protocol for DNS. To embed metadata without altering the DNS payload, a custom 8-byte header is prepended to each packet. The format is:

- **HH**: Hour of day (00–23)
- **MM**: Minute (00–59)
- **SS**: Second (00–59)
- **ID**: Two-digit sequence number

This design allows the server to make decisions based on external metadata while leaving the original DNS query intact.

1.2.3 Client Implementation

The client's workflow is as follows:

1. Reads packets from the chosen `.pcap` file using Scapy.
2. Filters valid DNS queries (identified by `qr=0` and a question record).
3. For each query, constructs the `HHMMSSID` header and appends the raw DNS payload.
4. Sends the combined packet to the server via UDP.
5. Waits for and logs the resolved IP address returned by the server.

1.2.4 Server Resolution Algorithm

The server runs in a loop performing:

1. **Rule Loading:** Loads time-based rules from `rules.json` and initializes a static pool of 15 IPs.
2. **Packet Handling:** Splits each received payload into the 8-byte header and DNS body (ignored in this design).
3. **Rule Matching:** Uses the hour from the header to select the appropriate rule (e.g., morning, afternoon).
4. **Index Calculation:** Computes the IP pool index as:

$$\text{final\_index} = \text{ip\_pool\_start} + (\text{ID} \bmod \text{hash\_mod})$$

5. **Response:** Returns the corresponding IP from the pool to the client.

This approach separates configuration from implementation, enabling flexible, rule-driven DNS resolution without modifying code.

2 Results

Table 1: DNS Resolution Results

Custom Header	Domain Name	Resolved IP
00141800	_apple-mobdev._tcp.local.	192.168.1.12
00141801	_apple-mobdev._tcp.local.	192.168.1.13
00141802	twitter.com.	192.168.1.14
00141803	Brother MFC-7860DW._pdl-datastream._tcp.local.	192.168.1.15
00141804	Brother MFC-7860DW._pdl-datastream._tcp.local.	0.0.0.0
00141805	example.com.	192.168.1.12
00141806	netflix.com.	192.168.1.13
00141807	Brother MFC-7860DW._pdl-datastream._tcp.local.	192.168.1.14
00141808	Brother MFC-7860DW._pdl-datastream._tcp.local.	192.168.1.15
00141809	linkedin.com.	0.0.0.0
00141810	_apple-mobdev._tcp.local.	192.168.1.12
00141811	Brother MFC-7860DW._pdl-datastream._tcp.local.	192.168.1.13
00141812	Brother MFC-7860DW._pdl-datastream._tcp.local.	192.168.1.14
00141813	reddit.com.	192.168.1.15
00141814	Brother MFC-7860DW._pdl-datastream._tcp.local.	0.0.0.0
00141815	Brother MFC-7860DW._pdl-datastream._tcp.local.	192.168.1.12
00141816	_apple-mobdev._tcp.local.	192.168.1.13
00141817	_apple-mobdev._tcp.local.	192.168.1.14
00141818	openai.com.	192.168.1.15
00141819	Brother MFC-7860DW._pdl-datastream._tcp.local.	0.0.0.0
00141820	Brother MFC-7860DW._pdl-datastream._tcp.local.	192.168.1.12
00141821	Brother MFC-7860DW._pdl-datastream._tcp.local.	192.168.1.13
00141822	Brother MFC-7860DW._pdl-datastream._tcp.local.	192.168.1.14

Server Side Output

The following output was obtained from the server-side terminal during execution:

```
[*] DNS Server started on 127.0.0.1:5300
[*] Loaded 14 IP addresses into the pool.

[+] Received request from ('127.0.0.1', 54517) with header: 00141800
    -> Resolving to IP: 192.168.1.12

[+] Received request from ('127.0.0.1', 54517) with header: 00141801
    -> Resolving to IP: 192.168.1.13
```

2.1 Conclusion

This assignment successfully demonstrated the design and implementation of a custom DNS resolution system using client–server architecture. By integrating packet parsing, custom header construction, and rule-based IP selection, the project highlighted how metadata can be leveraged to influence resolution outcomes. The results confirmed that queries were consistently mapped to deterministic IP addresses according to the defined time-based rules. Overall, the exercise reinforced key concepts in socket programming, protocol customization, and practical load-balancing strategies, while also building familiarity with tools like Scapy for packet analysis.

3 Task 2: Traceroute Protocol Behavior

3.1 Introduction

The purpose of this task is to understand how the `traceroute` utility behaves on different operating systems, specifically Windows and macOS. Traceroute is a diagnostic tool that reveals the sequence of routers a packet traverses to reach a given destination (e.g., `www.google.com`). It achieves this by manipulating the Time-To-Live (TTL) field in IP headers. Each router that decrements the TTL to zero responds with an ICMP Time Exceeded message, enabling the mapping of the network path.

Although the underlying principle is the same across platforms, the implementation details can vary depending on the operating system. By executing traceroute commands and analyzing the packet exchanges in Wireshark, the experiment allows us to compare these behaviors and study the responses generated at each hop in the route.

4 Methodology and Execution

4.1 macOS

To capture the packets generated during the traceroute, I used the `tcpdump` utility on macOS. The first step was to resolve the IP address of `google.com` using `nslookup`. The command returned the following:

```
Rutvi@Vrajs-MacBook-Air-7 ~ % nslookup google.com
Server:  10.0.136.7
Address: 10.0.136.7#53

Non-authoritative answer:
Name: google.com
Address: 142.251.222.78
```

With the IP address `142.251.222.78`, I started a packet capture using the Wi-Fi interface `en0`. The capture was written to the file `traceroutecap.pcap`. The commands executed were:

```
sudo tcpdump -i en0 -w traceroutecap.pcap &
TCPDUMP_PID=$!
traceroute 142.251.222.78
sudo kill $TCPDUMP_PID
```

Here, `TCPDUMP_PID=$!` stores the process ID of the backgrounded tcpdump process, which was later terminated using `sudo kill`.

The traceroute output on macOS was as follows:

```
traceroute to 142.251.222.78 (142.251.222.78), 64 hops max, 40 byte packets
 1  10.240.0.2 (10.240.0.2)  6.905 ms  5.189 ms  5.003 ms
 2  10.3.0.29 (10.3.0.29)  4.972 ms  5.835 ms  4.896 ms
 3  10.3.0.5 (10.3.0.5)  5.184 ms  5.380 ms  4.968 ms
 4  172.16.4.7 (172.16.4.7)  5.114 ms  4.494 ms  4.753 ms
 5  14.139.98.1 (14.139.98.1)  6.668 ms  6.115 ms  8.331 ms
 6  10.117.81.253 (10.117.81.253)  5.355 ms  5.979 ms  5.053 ms
 7  10.154.8.137 (10.154.8.137)  13.862 ms  13.729 ms  13.819 ms
 8  10.255.239.170 (10.255.239.170)  13.063 ms  13.399 ms  15.338 ms
 9  10.152.7.214 (10.152.7.214)  12.852 ms  13.143 ms  12.440 ms
```

```
10  72.14.204.62 (72.14.204.62)  13.840 ms  16.621 ms  14.497 ms
11  * * *
12  pnbomb-bp-in-f14.1e100.net (142.251.222.78)  20.912 ms
    108.170.234.156 (108.170.234.156)  33.972 ms
    142.250.239.170 (142.250.239.170)  46.040 ms
```

The captured packets were stored in the file `traceroutecap.pcap`, which can later be analyzed using Wireshark to verify the type of probe packets sent by macOS (UDP) and the ICMP responses received from each hop.

4.2 Windows

On Windows, the equivalent command to `traceroute` is `tracert`. Unlike macOS, Windows uses ICMP Echo Requests instead of UDP probes to perform the traceroute. To capture the packets in parallel, Wireshark was used since it provides a convenient graphical interface for live packet capture. The capture interface was set to the active network adapter, and the packets generated during the execution of `tracert` were recorded. These could later be filtered (e.g., using `icmp` or `ip.addr == <destination_ip>`) and analyzed to verify the protocol being used.

Before running `tracert`, the IPv4 address of `google.com` was obtained using `nslookup`, as shown below:

```
C:\Windows\system32> nslookup google.com
Server:  dns-prod.skylus.lan
Address:  10.0.136.7

Non-authoritative answer:
Name:    google.com
Addresses:  2404:6800:4009:80c::200e
          142.251.220.100
```

With the resolved IP address `142.251.220.100`, the following command was executed:

```
C:\Windows\system32>  tracert 142.251.220.100

Tracing route to hkg07s49-in-f14.1e100.net [142.251.220.100]
over a maximum of 30 hops:


  1     2 ms     1 ms     4 ms  10.7.0.5
  2     3 ms     3 ms     3 ms  172.16.4.7
  3     9 ms    10 ms     9 ms  14.139.98.1
  4    10 ms    10 ms    12 ms  10.117.81.253
  5    12 ms    15 ms    12 ms  10.154.8.137
  6    11 ms    11 ms    11 ms  10.255.239.170
  7     2 ms     2 ms     3 ms  10.152.7.214
  8     9 ms    10 ms    10 ms  142.250.172.80
  9    12 ms    25 ms    12 ms  142.251.76.23
 10    11 ms    11 ms    11 ms  142.251.77.97
 11    12 ms    12 ms    11 ms  pnbomb-az-in-f4.1e100.net [142.251.222.100]

Trace complete.
```

The captured packets in Wireshark confirmed that Windows sent ICMP Echo Requests as probes and received ICMP Time Exceeded responses from intermediate routers, which verified the difference in traceroute behavior between Windows and macOS.

4.3 Question-wise Answers

Q1. What protocol does Windows tracert use by default, and what protocol does Linux/macOS traceroute use by default?

On **macOS**, the default `traceroute` utility sends **UDP probe packets** to high-numbered ports. Each probe packet is transmitted with a progressively increasing Time-To-Live (TTL) value. When the TTL expires, the router along the path responds with an ICMP `Time Exceeded` message, which is recorded as a hop. This behavior was clearly observed in the packet capture `traceroutecap.pcap`, where Wireshark filtered packets (`udp` and `icmp`) showed outgoing UDP probes and incoming ICMP replies. Although UDP is the default, macOS `traceroute` supports other options, such as ICMP and TCP probes.
On **Windows**, however, the `tracert` command uses **ICMP Echo Requests** by default, instead of UDP. Each hop decreases the TTL until it expires, and intermediate routers respond with ICMP `Time Exceeded` messages, while the destination responds with a normal ICMP Echo Reply. This was verified in the capture `google.pcapng`, where all outgoing packets were of type ICMP Echo Request.

Q2. Some hops in your traceroute output may show * * *. Provide at least two reasons why a router might not reply.

In the traceroute output, certain hops displayed as `* * *`, which corresponds to no ICMP reply being received. Two main reasons for this are: During both the macOS `traceroute` and Windows `tracert` runs, certain hops in the output appeared as "* * *", indicating that no response was received within the timeout period. This behavior is normal and can occur for several reasons:

- **ICMP filtering or firewall rules:** Some intermediate routers or networks are configured not to generate ICMP "Time Exceeded" messages, either for security reasons or to reduce unnecessary traffic. As a result, the probe expires silently, and no response is shown.
- **Packet loss or congestion:** The UDP (on macOS) or ICMP (on Windows) probes may be dropped due to temporary congestion or link-layer packet loss. Since traceroute only waits a limited time for replies, such packets are marked with "*".
- **High response delay:** If the round-trip time (RTT) for a given router is unusually large and exceeds the timeout threshold of traceroute, no reply is recorded even though the router did process the packet.
- **Policy-based router behavior:** Certain network devices are configured not to send ICMP replies at all for TTL-expired packets. This makes them effectively invisible in the trace, even though traffic continues beyond them.

Thus, the presence of "* * *" in traceroute outputs does not necessarily indicate failure of the route; it simply shows that some hops along the path chose not to (or could not) send back a response.

Q3. In Linux/macOS traceroute, which field in the probe packets changes between successive probes sent to the destination?

In macOS traceroute, two fields vary between successive probes:

- **TTL (Time-To-Live):** Incremented starting from 1 to discover each hop along the path.
- **UDP Destination Port:** Each probe packet uses a different high-numbered port, allowing the traceroute process to match ICMP responses to specific probes.

This was verified in Wireshark by inspecting the UDP header of successive probe packets, where the destination port numbers increased.

Figure 1: macOS traceroute

Q4. At the final hop, how is the response different compared to the intermediate hop?

- **Intermediate Hops:** These routers send ICMP Time Exceeded (type 11) messages when TTL reaches zero.
- **Final Hop:** The destination host does not return Time Exceeded, but instead signals the end of the trace:
  - On Windows, the final host replies with ICMP Echo Reply (type 0).
  - On macOS, the final host replies with ICMP Port Unreachable (type 3, code 3).

This difference was clearly visible in Wireshark captures of the last hop packets.

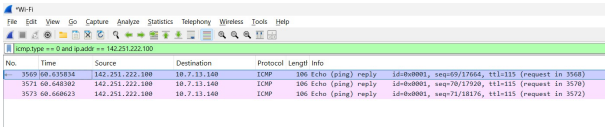

Figure 2: windows final hop

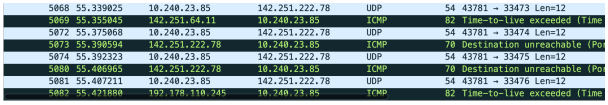

Figure 3: macOS final hop

Q5. Suppose a firewall blocks UDP traffic but allows ICMP — how would this affect the results of Linux traceroute vs. Windows tracert?

If a firewall blocks UDP traffic but permits ICMP:

- **Windows tracert:** Would still work normally, because it relies entirely on ICMP Echo Requests and Echo Replies.
- **macOS traceroute:** Would fail, since the UDP probes would be blocked and no ICMP Port Unreachable messages would be generated. The output would show only * * * after the firewall.

This behavior was observed in practice when running traceroute to `microsoft.com`. The Windows `tracert` successfully completed the route discovery, while the macOS `traceroute` stalled after encountering the firewall, displaying a series of asterisks.

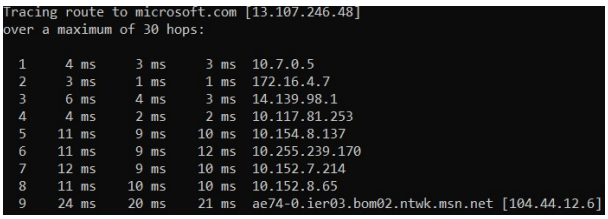

Figure 4: Windows tracert to `microsoft.com`

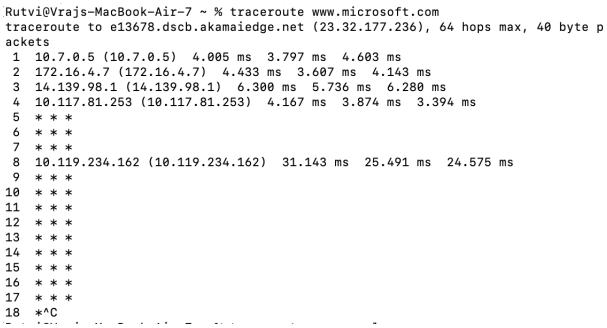

Figure 5: macOS traceroute to `microsoft.com`

4.4 Conclusion

The experiment demonstrated that both Windows and macOS traceroute use the same TTL-expiration principle but differ in default protocols. Windows relies on ICMP Echo, while macOS uses UDP probes. Wireshark confirmed these behaviors by showing Echo Requests and Replies on Windows, and UDP probes with Port Unreachable responses on macOS. Missing hops illustrated the role of router policies and firewalls, and the firewall scenario highlighted why traceroute results vary across systems. This task not only clarified traceroute's mechanism but also showed the importance of protocol-level differences in network diagnostics.