Assignment 1

CS331: Computer Networks

Siddhesh Umarjee (23110347) Shounak Ranade (23110304)

# task 1: Custom header and dns resolver

## Introduction

Task 1 implements a lightweight client–server DNS resolver. It helps us understand packet parsing, header design, and policy-driven IP routing. The client filters DNS queries from a given PCAP, prepends an 8-byte custom header “HHMMSSID” carrying a UTC timestamp and sequential ID, and sends the message unchanged otherwise; the server parses the header, extracts the domain from the original DNS payload, and applies time-based routing with a 15-IP pool by selecting a 5-address segment based on hour and choosing a specific address via ID modulo, finally returning and logging the resolved address alongside the original query.

We use the language Go for this task.

## Prerequisite

The GitHub repository can be found here: <https://github.com/IdkRandomTry/DNS-Resolver>. The README.md file has the necessary instructions to run the code. Go 1.21+ recommended

Required Packages:

* Standard library: net (UDP sockets), time (UTC header), fmt/log/os/bufio/flag/strconv.
* github.com/google/gopacket and github.com/google/gopacket/pcap for PCAP reading and packet iteration.
* github.com/google/gopacket/layers for DNS layer extraction.
* golang.org/x/net/dns/dnsmessage for safe DNS question parsing on the server.

## Methodology

We maintain a folder structure as follows:

dns\_resolver/

- client/

  - client.go       # Client code

  - 1.pcap           # PCAP file

- server/

  - extract\_domain.go # Extract domain name

- server.go       # UDP server

  - ip\_select.go     # IP selection logic

- .gitignore        # Recommended ignore rules

- go.mod            # Go module file

### Client

The client takes a PCAP (Packet Capture), iterates packets, and filters DNS queries, ensuring responses (QR=1) are skipped to avoid duplication. For each query, it constructs an 8-byte custom header “HHMMSSID” using UTC time and a (zero-padded) sequential ID. It then prefixes this header to the untouched DNS payload and transmits the resulting datagram via UDP to the server endpoint. The client maintains a simple, sequential ID counter modulo 100 to bound header size. It also logs the timestamp, ID, and send status for traceability during test runs.

Some important Go calls used:

* net.ResolveUDPAddr: Parses “host:port” into a UDP endpoint structure.
* net.DialUDP: Dials a UDP “connection” to a remote UDPAddr, returning a UDPConn that writes to the implicit remote address.
* (\*net.UDPConn).Write: Writes a byte slice to the connected UDP peer; returns number of bytes written and error.
* pcap.OpenOffline: Opens a saved pcap file for offline reading and returns a handle for packet access.
* (\*pcap.Handle).LinkType: Returns the link-layer type of the capture source (e.g., Ethernet), required by NewPacketSource.
* gopacket.NewPacketSource: Wraps a PacketDataSource (e.g., pcap handle) and decoder to stream decoded Packet instances.
* (\*gopacket.PacketSource).Packets: Returns a channel that asynchronously yields decoded packets until EOF or error.
* gopacket.Layer: Retrieves a specific decoded layer from a Packet by its LayerType; returns nil if absent.

### Server

The server listens on a UDP socket on port 1053 (stand-in for port 53). It extracts the DNS payload and slices the first 8 bytes as the custom header. The remaining bytes are parsed as a DNS message to extract the queried domain name using the function defined in `extract\_domain.go`. The server the applies time-based routing by mapping the hour to one of three 5-IP pool segments (morning/afternoon/night) and selects an IP using ID % 5 to index within that segment, thus implementing deterministic DNS-style load distribution. The function used is defined in `ip\_select.go`

Some important Go calls used

* net.UDPAddr: Represents a UDP endpoint with IP and Port fields. Used to specify bind addresses.
* net.ListenUDP: Creates a UDP socket bound to the given local address; if IP is unspecified (nil/0.0.0.0), it listens on all local addresses. Returns a \*net.UDPConn.
* (\*net.UDPConn).ReadFromUDP: Reads a UDP datagram, returning the count, source address, and error; blocks until data is available or an error occurs.
* dnsmessage.Parser: Stateful parser for incrementally decoding DNS messages from a byte slice.
* (\*dnsmessage.Parser).Start: Parses the DNS header and prepares the parser to read the Question section. Returns the parsed Header and any error.
* (\*dnsmessage.Parser).Question: Parses and returns the next Question; returns ErrSectionDone when no more questions remain.
* (dnsmessage.Name).String: Formats the domain name in presentation form (e.g. “[www.example.com](http://www.example.com.)”)

## Results

A computer screen with white text

AI-generated content may be incorrect.A computer screen with white text

AI-generated content may be incorrect.A screen shot of a computer

AI-generated content may be incorrect.Here are results from running the code at different times. (Adjusted time manually for testing)

## Conclusion

This toy server-client DNS resolver helped us gain insight into practical DNS handling over UDP. The current design included parsing real query payloads, designing and carrying a compact “HHMMSSID” header, and enforcing deterministic, time-based load selection across a fixed IP pool. It clarified how transport choices, header design, and stateless routing rules interact with each other.

# task 2: Traceroute Protocol Behavior

## Introduction

Task 2 explores the underlying mechanisms and differences in protocol behaviour of the traceroute utility across various operating systems, namely Windows and Linux. This gives an insight into the network protocols employed by the operating systems. We also compare the differences observed on Wireshark and attempt to give reasoning for them. We set our destination to [www.google.com](http://www.google.com).

## Tools Used

The tools used for this task include the following:

* Wireshark: captures and analyzes network traffic.
* `traceroute` function to trace network routes on Linux.
* `tracert` function to trace network routes on Windows.
* A screen shot of a computer

  AI-generated content may be incorrect.To test Linux network protocols, WSL 2.5 was used, which includes a Linux kernel network backend. Checking ipconfig on Windows shows a virtual Ethernet connection to WSL, supporting our claim that running traceroute on WSL produces results identical to running it on a native Linux system.

## Methodology

To study the behaviour of the traceroute utility across different operating systems, we conducted experiments on both Windows (using the tracert command) and Linux (using the traceroute command inside WSL2). During execution, network traffic was captured using Wireshark on the Windows host machine to observe the probe packets and responses. We applied protocol filters (icmp for Windows, udp or icmp for Linux) to isolate relevant traffic and identify differences in protocol usage, packet structure, and response behavior at intermediate and final hops. The captured data was then analyzed to explain the questions asked in the assignment.

## Results

Using the data we observed in Wireshark, we attempt to answer the following questions.

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

ANSWER:

A screenshot of a computer screen

AI-generated content may be incorrect.**Windows tracert**: uses **ICMP Echo Requests** (just like ping) with increasing time to live for each subsequent request. (notice the increasing ttl in screenshot)

A screenshot of a computer screen

AI-generated content may be incorrect.**Linux traceroute**: uses **UDP datagrams to high, unused port numbers** (>33434) which are initially responded with by ICMP “Time Exceeded”. It also relies on increasing ttl to find the port.

1. A screenshot of a computer screen

   AI-generated content may be incorrect.Some hops in your traceroute output may show \*\*\*. Provide at least two reasons why a router might not reply.

**Firewall**: ICMP or UDP traffic may be blocked at intermediate hops.

**Overloading**: if a router is too busy or overloaded, it may simply drop our probe packets and not generate ICMP “Time Exceeded” replies.

**ICMP rate limiting**: some routers intentionally drop or throttle ICMP “Time Exceeded” messages to avoid being overwhelmed.

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

The **TTL (Time To Live)** field is incremented with each probe.

We also observe the **UDP destination port number** changes for each probe. This lets traceroute match “ICMP Port Unreachable” replies back to the right probe.A screenshot of a computer program

AI-generated content may be incorrect.A computer screen shot of a program

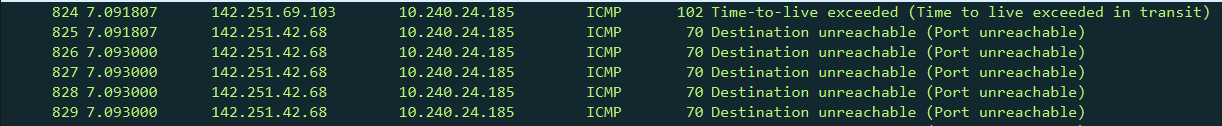
AI-generated content may be incorrect.A screenshot of a computer screen

AI-generated content may be incorrect.

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

**Intermediate hops**: receive **ICMP Time Exceeded** messages when TTL expires.

**Final hop** (destination):

In **Linux UDP traceroute**: destination replies with **ICMP Port Unreachable** (since no service is listening on the high UDP port).

A screen shot of a computer

AI-generated content may be incorrect.In **Windows ICMP tracert**: destination replies with an **ICMP Echo Reply**.

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

**Linux traceroute** is UDP-based and hence would fail, since its probes are UDP packets.

**Windows tracert** is ICMP-based and hence would still work normally, since it uses ICMP Echo Requests and expects ICMP replies.

## Conclusion

Traceroute works by sending probe packets with gradually increasing **TTL values**. This causes routers along the path to reply with ICMP *Time Exceeded* messages. Although this core concept is same in Linux and Windows trace route implementation, they differ in the following way: On Linux, these probes are typically **UDP packets** while on Windows, they are **ICMP (ping) requests**.