

ASSIGNMENT 1

Computer Networks

Tejas Lohia, Umang Shikarvar

23110335, 23110301

Contents

1	DNS Resolver	2
1.1	Introduction	2
1.1.1	DNS:	2
1.1.2	PCAP files:	2
1.1.3	DNS resolver:	2
1.2	Tools	3
1.3	Methodology and System Design	3
1.3.1	Main () function of client	4
1.3.2	Packet structures and global pointers	10
1.3.3	DNS_custom_header	14
1.3.4	Functions	14
1.4	DNS resolution server implementation	24
1.5	Logged IP Addresses.	26
1.6	Observations	27
2	Traceroute Protocol Behavior	31

Chapter 1

DNS Resolver

1.1 Introduction

1.1.1 DNS:

The domain server name is one of the most critical protocols in computer Networks. It is highly preferable for a user to input human-friendly domain names such as `www.google.com`. However, to make these interpretable to the networking protocols to avail services, a system is required which would convert human-friendly domain names to machine-level IP Addresses. DNS acts as a distributed and heirarchical translator, making them very fundamental in the computer networks as it would be very difficult for users to remember numerical IP Addresses. DNS provides a seamless service to the clients to get the IP Addresses using a heirarchical server presence. Citing the central role, DNS is an important subject of study in research in computer networks.

1.1.2 PCAP files:

PCAP files are critical in the practical network analysis. These files contain packet datas that have been captured from interfaces of live networks using tools like `tcpdump` and `wireshark`. PCAP files store packet data in sequential order which allows users to study and dissect network traffic post capture, without relying on the active network connection. This provides an environment to study real time packets instead of working with artificially generated packets.

1.1.3 DNS resolver:

DNS resolver is a network service responsible for the tranlation by acting as an intermediary between client such as web browser and DNS system. When there is a DNS query, it is handled and resolved either by returning the cache if exists or by requesting from the DNS servers.

1.2 Tools

- **Programming Language:** C language and Python 3.12.9
- **Editor/IDE:** Visual Studio Code — Used for coding, debugging and execution.
- **Version Control:** Git and GitHub — Used to track and the changes in the code, and to improve maintainability of the codebases.
- **Virtual Environment:** `assign1` — To prevent library version conflicts by isolating working environments.
- **wireshark:** — To verify the results obtained by the codes over PCAP files.

1.3 Methodology and System Design

The overall workflow adopted for this project was to analyze the DNS data from packet capture files. The file contained data for multiple protocols, out of which only entries corresponding for DNS queries were extracted. The client file reads packets from .pcap file, parses them to extract the DNS data from thses packets. The data is concatenated with custom data of 8 bytes containing information about custom header containing time and `sequence_id` and is passed to the server. The server in turn returns the domain name from the DNS data passed to it.

Code is written at a low level without actually using the pcap library. Code parses each of the packet, filters based on the type and then slices the network layer packets to obtain the raw data.

At high level the code follows the conventional client-server model. Client sends the DNS queries and server returns the IP address over UDP protocol which is preferrably used for DNS protocol.

1.3.1 Main () function of client

```
1 int main(int argc, char *argv[]) {
2
3     // if (argc != 2) {
4     //     printf("Usage: %s <pcap file>\n", argv[0]);
5     //     return 1;
6     // }
7
8     printf("PCAP Client\n");
9     //char *filename = argv[1];
10    //char *filename = "./p.pcap";
11
12    // Set the pcap file to be processed
13    char *filename = "./6.pcap";
14
15    //file which would be used to analyze the DNS packets
16    char *dnsFileName="./dns.txt";
17
18    dnsFile = fopen(dnsFileName, "w");
19
20    dnsReportFp = fopen("./dnsReport.txt", "w");
21
22
23    if (!dnsReportFp) {
24        printf("Failed to open file\n");
25        return -1;
26    }
27
28    /*
29    Opening the pcap file in the binary format which is stored in the
30    little Endian format in the memory.
31    */
32
33    FILE *fp = fopen(filename, "rb");
34
35    if (!fp) {
36        printf("Failed to open file\n");
37        return -1;
38    }
```

Listing 1.1: Opening the files

Opening files

This part of the code opens 6pcap files in binary format as **fp** which is stored in binary format in Little Endian. dnsReport file is opened to store the required content which stores CustomHeaderFile, Domainname and resolved IP Addresses. Another file dns.txt is opened to store the DNS data extracted from the packets to debug.

```

1      create_dns_client_socket(); //we initialize the client socket and
2          configures the server_address
3
4      pcap_hdr_t header; //will store the header of the pcap file
5
6      /*
7      pcap file opened in binary format
8      fread : C library call to read the binary data.
9      It copies the first sizeof(header) amount of content to the structure.
10     This structure is the header of the pcap file
11     */
12
13     size_t read_bytes = fread(&header, 1, sizeof(header), fp);
14     if (read_bytes != sizeof(header)) {
15         printf("Failed to read pcap header\n");
16         fclose(fp);
17         return -1;
18     }
19
20     //printing the header of the pcap file
21     print_global_header(&header);
22
23     /*
24
25     Magic number is used to validate the type of the file.
26     Magic number : 0xa1b2c3d4 confirms pcap file
27     */
28
29     if(header.magic_number != 0xa1b2c3d4){
30         printf("%s is not valid pcap file\n", argv[1]);
31         exit(-1);
32     }

```

Listing 1.2: Creating the socket and checking the validity of the pcap file

Socket Initialization

This part of the code calls the `create_dns_client_socket` function which creates the socket for client and initializes server address. A structure is initialized of the type `pcap_header`, which is used to copy the pcap header data from the pcap file. Pointer to this structure is passed to a library system call `fread` to obtain the data, which equals the size of the of the structure. Another function `print_global_header` is called to print the header information extracted from the pcap file while incrementing the file read pointer.

Magic Number: They are used to check the type of the opened file. `0xa1b2c3d4` is the magic number for the pcap filetype. Code exists in case the magic number obtained does

not match the xalb2c3d4, as it indicates improper file type.

```
1
2 fprintf(dnsReportFp, "\tCustomHeaderFile\tDomainname\t\tResolved IP
   Address\n");
3     fprintf(dnsReportFp, "\t    (HHMMSSID)\n\n");
4
5     while(len > 0){
6         record_packet1=NULL;
7         printf("packet no %d\n",record_num++);
8
9     /*
10
11     read_pcap_packet is called, which fetches the length of the packet.
12     Importantly it also allocates memory to the packet read and
13     record_packet1 points to the array where packet is stored.
14
15     */
16
17     len=read_pcap_packet(fp,&record_packet1);
18
19     if(len == 0 ){
20         printf("end of file\n"); //len
21         break;
22     }
23
24     if(len == -1){
25         printf("Error in the reading the packet\n");
26         printf("Breaking the loop\n");
27         break;
28     }
29
30     //printf("address %p",(void *)record_packet1);
31     //printf("Packet data (first 32 bytes or less):\n");
32     //for (uint32_t i = 0; i < 32; i++) {
33     //    printf("%02x ", *(record_packet1+i));
34     //}
35     //printf("\n");
36     //ethernet_header_t *eth=(ethernet_header_t *)record_packet1;
37     //printf("ethernet type %x\n",eth->ethertype);
38     //if (ntohs(eth->ethertype) != 0x0800) {
39     //    printf("Not an IPv4 packet\n\n");
40     //    return (-1);
41     //}
42
43     //getting the offset in the packet to point at the DNS data
44
45     dns_offset=parse_pcap_packet(record_packet1,len);
```

```
46     dns_packet_size=len-dns_offset;
47     dns_custom_packet_size= dns_packet_size+8;
```

Listing 1.3: Iterating over the pcap file

Iteration over all the packets

This part of the code iterates through all the packets in the pcap file.

record_packet1 is a pointer to an array of characters which will point to the array storing the each packet data.

Function read_pcap_packet function is called with over the pointer to the current packet in the file and the pointer to store the data.

if the function returns 0, code exits as it has reached the end of the pcap file, else if the function returns -1, then there was some error in reading packet and the loop breaks.

From the array containing the packet data, to extract the DNS data we are required to find the offset to reach the DNS data.

Function parse_pcap_packet is invoked to find the offset in that array.

As the custom added header would be of 8 bytes, the size of the final packet would be **length of the original packet - dns offset** which gives the length of the DNS data + 8, which is the size of the custom header added.

```
1
2  if(dns_offset > 0) // Meaning DNS packet
3      {
4          //printf("DNS packet\n");
5          //fprintf(dnsFile,"DNS offset %d %d\n",dns_offset,len);
6          //for(int i=0;i< (dns_packet_size);i++)
7          //    fprintf(dnsFile,"%02x ", *(record_packet1+i+
8              dns_offset));
9
10         //dns_name extracts the human readable domain name by
11         //accessing the DNS data in the packet.
12         dns_name((record_packet1+dns_offset+12),dn_query_name);
13
14         fprintf(dnsFile,"\n Total Bytes %d : %s",
15             dns_custom_packet_size,dn_query_name);
16
17         //custom_dns_packet structure stores the final packet
18         //containing the custom data header along with the DNS
19         //data.
```



```

16         make_custom_packet(&custom_dns_packet,(record_packet1+
17                               dns_offset),dns_packet_size);
18
19         // custom_dns_packet is sent to the server to get the ipv4
20         // address.
21         int bytesSent=send_dns_msg_to_server(custom_dns_packet,
22             dns_custom_packet_size);
23         fprintf(dnsFile,"\n Bytes Sent %d : ",bytesSent);
24         for(int i=0;i< (dns_packet_size+8);i++)
25             fprintf(dnsFile,"%02x ", *(custom_dns_packet+i));
26         //free(custom_dns_packet);
27         //usleep(100000);
28
29         //dns_reply_ip is contains the received IP address from the
30         // server.
31         receive_dns_msg_from_server(dns_reply_ip,10);
32
33         // custom_dns_packet is type casted to the structure
34         // storing the custom header data.
35         dns_custom_header_t *dns_ch=(dns_custom_header_t*)(
36             custom_dns_packet);
37
38         //final data is stored to the dnsReport
39         fprintf(dnsReportFp,"%t%02d%02d%02d%02d\t\t%s\t\t%s\n",
40             (int)dns_ch->hour,
41             (int)dns_ch->min,
42             (int)dns_ch->sec,
43             (int)dns_ch->seq_no,
44             dn_query_name,
45             dns_reply_ip
46
47             );
48         free(custom_dns_packet);
49     }
50     //printf("packet no %d\n",record_num++);
51     if(record_packet1 != NULL)
52         free(record_packet1);
53 }

```

Listing 1.4: Processing the DNS packet, DNS querying and storing the results

Custom header addition, querying and receiving IPv4 address

This is the critical part of the code which extracts the name from the packet data, queries to the DNS server and stores the received IPv4 address into the dnsReport file.

dns_name function is invoked which passes the pointer to the final location where the DNS data starts in the record_packet1 array and copies the human understandable name

to the variable `dn_query_name`.

`make_custom_packet` function is called which combines the custom header data with the DNS data in the packet to store the final DNS query packet in variable `custom_dns_packet`.

`send_dns_msg_to_server` function is invoked which finally sends the DNS query.

`receive_dns_msg_from_server` is invoked which receives the ip address reply from the server, and finally the data is stored in `dnsReport`.

1.3.2 Packet structures and global pointers

```
1 int sockfd;
2 struct sockaddr_in server_addr;
3
4 FILE *dnsFile;
5 FILE *dnsReportFp;
```

Listing 1.5: socket, server_addr and file pointers

socket stores the socket number.

server_addr stores the information about the server IP, port and protocol over which the connection would be laid.

dnsFile and dnsReportFp are pointers to text files.

```
1 /*
2 pcap file structure:
3     24 bytes pcap header
4     16 bytes record header
5     incl_len bytes record or packet (Can be for any type)
6     (this repeats)
7 */
8
9 // 24 Bytes for PCAP Header
10 typedef struct pcap_hdr_s {
11
12     uint32_t magic_number;    /* magic number */
13     uint16_t version_major;  /* major version number */
14     uint16_t version_minor;  /* minor version number */
15     uint32_t thiszone;       /* GMT to local correction */
16     uint32_t sigfigs;        /* accuracy of timestamps */
17     uint32_t snaplen;        /* max length of captured packets, in
18                             octets */
19     uint32_t network;        /* data link type */
20 } pcap_hdr_t;
```

Listing 1.6: Pcap file structure and pcap file header

Understanding the pcap file structure

pcap file contains 24 bytes of header data. Next 16 bytes are the header for a packet, followed by packet with variable length.

For every packet of variable length, there is a header of 16 bytes. So, there are alternating packet headers and packets.

Understanding the pcap header structure

Importantly pcap header file contains the magic number which is used to check whether the opened file is pcap file.

This is followed by version umbers, offset to GMT, accuracy of the timestamps, max length of the captured packets and data link type.

[Ref: Libpcap File Format Overview \(Wireshark Wiki\)](#)

Packet headers

```
1  /*
2  record header structure:
3      This is a 16byte header before every record in the PCAP file.
4      Third entry of this header gives information about the length of
5          incl_len
6      While forth entry gives the original length of the packet, in case
7          cropped
8  */
9
10 typedef struct pcaprec_hdr_s {
11     uint32_t ts_sec;           /* timestamp seconds */
12     uint32_t ts_usec;         /* timestamp microseconds */
13     uint32_t incl_len;        /* number of octets of packet saved in
14         file */
15     uint32_t orig_len;        /* actual length of packet */
16 } pcaprec_hdr_t;
```

Listing 1.7: Packet headers

Two initial fields store the seconds and microseconds of the time packet received since the UNIX epoch.

Packets are of varied length, therefore `incl_len` stores the length of the packet, while `orig_len` stores the length of the original packet in cases where the packet has been sliced to some max length.

[Ref: Libpcap File Format Overview \(Wireshark Wiki\)](#)

Packet Structures

Record/Packet structure

- 14 bytes: Ethernet Header
- IPv4 Header (typically 20 bytes, but can be more with options)
- 8 bytes: UDP Header *or* variable size for TCP Header / other protocol header
- 12 bytes: DNS Header *or* other protocol header
- DNS Data / other data in packet

Each of the packet contains a series of header data, and then the final data in the packet.

1. First header is the **Ethernet header** which is of 14 bytes containing 3 fields.

Ethernet Structures

```
1 typedef struct ethernet_header_s{
2     uint8_t dest_mac[6];
3     uint8_t src_mac[6];
4     uint16_t ethertype;
5 } ethernet_header_t;
```

Listing 1.8: Ethernet Header Structure

dest_mac contains the destination mac address.

src_mac contains the source mac address.

ethertype is a 2 byte field storing the protocol the payload carries.

2. Next header is the **IPv4 header** which is of typically 20 bytes but might vary.

IPv4 Structures

```
1 // IPv4 header structure
2 typedef struct ipv4_header_s {
3     uint8_t version_ihl;
4     uint8_t tos;
5     uint16_t total_length;
6     uint16_t identification;
7     uint16_t flags_frag_offset;
8     uint8_t ttl;
9     uint8_t protocol; //protocol of the entry
10    uint16_t header_checksum;
11    uint32_t src_ip;
12    uint32_t dest_ip;
13 } ipv4_header_t;
```

Listing 1.9: IPv4 Header Structure

It contains several fields:

- **version_ihl:** gives the sum of version and internet header length.
- **tos:** Used to indicate priority and handling instructions for routers.
- **total length:** Size of the entire packet (header + data) in bytes.
- **identification:** A unique number assigned to each packet sent by a host.
- **flags_frag_offset:** Used for packet fragmentation and reassembly.
- **ttl:** Counter to handle maximum number of hopping at every router.
- **protocol:** Indicates the protocol, where 17 is for UDP which is used later in the code.
- **header_checksum:** Detects corruption in transit.
- **src_ip and dest_ip:** IP Addresses of source and destination.

3. Next headers are **UDP header** which is of typically 8 bytes and **DNS header** which is of typically 12 bytes

UDP Structures

```
1 typedef struct udp_header_s {  
2     uint16_t src_port;  
3     uint16_t dest_port; //this should be 53 for DNS.  
4     uint16_t length;  
5     uint16_t checksum;  
6 } udp_header_t;
```

Listing 1.10: UDP Header Structure

UDP structure holds very important fields.

- **src_port:** Stores the port number of the source.
- **dest_port:** Stores the port number of the destination.
- **length:** Length of UDP header + data
- **checksum:** Error checking of header + data

dns_header structure

```
1 typedef struct dns_header_s {  
2     uint16_t transaction_id;  
3     uint16_t flags;  
4     uint16_t questions;  
5     uint16_t answer_rrs;  
6     uint16_t authority_rrs;  
7     uint16_t additional_rrs;  
8 } dns_header_t;
```

Listing 1.11: DNS Header Structure

The DNS header contains critical information about the DNS query and response. Length : 12 bytes

- **transaction_id:** A unique identifier set by the client. It helps the client match responses with queries.
- **flags:** A 16-bit field containing multiple control bits such as query/response (QR), opcode, authoritative answer (AA), truncated message (TC), recursion desired (RD), recursion available (RA), and response code (RCODE).
- **questions:** Specifies the number of entries in the Question Section.
- **answer_rrs:** Number of resource records in the Answer Section.
- **authority_rrs:** Number of resource records in the Authority Section.
- **additional_rrs:** Number of resource records in the Additional Section.

1.3.3 DNS_custom_header

We define a structure for custom header that would be appended at the beginning of all the DNS packets.

```
1 typedef struct dns_custeum_header_s {  
2     uint16_t hour;  
3     uint16_t min;  
4     uint16_t sec;  
5     uint16_t seq_no;  
6 } dns_custom_header_t;
```

Listing 1.12: custom DNS header structure

It stores 3 fields related to hour, minutes and time and forth field which is the sequence number.

Functions are invoked which capture the time in the device and also the sequence ID of the DNS packet.

1.3.4 Functions

create__dns_client_socket

All the applications need some gateway to send and receive data to the internet. Applications use sockets to send and receive the packet to OS which later communicates.

```

1 int create_dns_client_socket()
2 {
3     if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
4         printf("socket failed");
5         return (-1);
6     }
7     memset(&server_addr, 0, sizeof(server_addr));
8     server_addr.sin_family = AF_INET; //this is a standard practice to
        assign the value '2' for UDP and TCP
9     server_addr.sin_port = htons(12345); // Server port -> Ethernet is
        Big Endian and if the machine is little Endian, it converts the
        format for compativility
10    server_addr.sin_addr.s_addr = inet_addr("127.0.0.1"); // Server IP
        -> This is a special IP address, which is the self IP address,
        as server is also hosted on the same machine for this assignment
11    return(1);
12
13 }

```

Listing 1.13: create_dns_client_socket

sockfd is assigned the socket number using the socket systems call. In the function call we pass `AF_INET = 2` which is for TCPUDP protocol and `SOCK_DGRAM` which is used for UDP procotol

`AF_INET = 2` tells the socket to use IPv4 addresses.

`SOCK_DGRAM` Specifies the socket type. `SOCK_DGRAM` means datagram socket, which uses UDP at the transport layer.

`memset` is another important system's call which sets all the fields or the memory in the `server_addr` structure to be zero.

`sin_family` is assigned the value 2, which corresponds to `AF_INET`. This is the standard value used to indicate that the socket will use the IPv4 address family for communication.

`sin_port` field is initialized the value 12345 which is the server port. Critical: **htons:** This was critical before passing the value 12345, as the machines that we use are Little Endian, while the networking works on Big Endian convention. Thus `htons` is required to handle the conversion.

`s_addr` field is assigned the IP address of "127.0.0.1", which is basically the IP address to the same device. As this is for the port, it is fixed and does not change.

read_pcap_packet

This function is used to get the length of the packet. It performs another critical task of copying the content from the pcap file to an array.

```
1
2 int read_pcap_packet(FILE *fp, unsigned char **record_packet)
3 {
4     pcaprec_hdr_t record_hdr;
5     // Read the next record header
6
7     int len = fread(&record_hdr, 1, sizeof(record_hdr), fp);
8     if(len < 1){
9         printf("End of file\n");
10        return(0);
11    }
12
13    if (len != sizeof(record_hdr)) {
14        printf("Failed to read record header %d\n", len);
15        return (-1);
16    }
17
18    // Print the record header fields
19    //printf("Timestamp: %u.%06u seconds\n", record_hdr.ts_sec,
20        record_hdr.ts_usec);
21    //printf("Captured Length: %u bytes\n", record_hdr.incl_len);
22    //printf("Original Length: %u bytes\n", record_hdr.orig_len);
23
24    /*
25    incl_len: Field type in the record header struct
26    Allocation of char array with size being incl_len + 10 bytes extra
27
28    */
29
30    *record_packet = malloc(sizeof(char) * record_hdr.incl_len + 10); //
31        allocated 10 bytes extra
32    if( *record_packet == NULL){
33        printf("Memory allocation failed for record_packet\n");
34        return(-1);
35    }
36
37    /*
38    len stores the length of the packet
39    if len == 0:
40        end of file
41    else if len != incl_len:
42        error
```

```

43     else :
44         packet read correctly and size is returned
45
46 */
47
48     len =fread(*record_packet, 1, record_hdr.incl_len, fp);
49     if(len < 1){
50         printf("End of file\n");
51         return(0);
52     }
53     if (len != record_hdr.incl_len) {
54         printf("Failed to read record packet read %d expected %d\n",
55             len,record_hdr.incl_len);
56         return(-1);
57     }
58     return(record_hdr.incl_len);
59
60 }

```

Listing 1.14: Finding the length of the packet

Code Explanation:

Code defines a variable `record_hdr` which is of the `pcaprec_hdr` which would be storing the length of the packet.

From the pointer pointing to the file, content of length `record_hdr` is copied to `record_hdr`. Now `record_hdr` contains the header of the packet. `incl_len` field of this `record_hdr` stores the length of the packet (memory located).

In case `len < 1`, it denotes the end of the file thus returning zero to make the while loop exit in the main function of the code.

`record_packet` is a pointer to an array storing the character. To this pointer we allocate the space equal to the memory of packet + 10 (for safety).

Again the `fread` function is called over the pointer to the pcap file, to read the next `incl_len` which contains the packet data in the `fp` file.

This `incl_len` is returned by the function if packet data successfully copied.

`parse_pcap_packet()`

This function traverses through the packet, applies filter to classify into DNS packets. If the packet is a DNS packet, then returns the offset for DNS in that packet array.

```

1 int parse_pcap_packet(unsigned char *pcap_packet,int pcap_packet_len)
2 {
3     // Parse Ethernet header

```

```

4 //printf("Parsing\n");
5 int eth_len=14;
6 int ipv4_min_len=20;
7 int udp_len=8;
8
9 if (pcap_packet_len < eth_len) {
10     printf("Packet Ethernet header short pcap_packet_len %d\n\n",
11         pcap_packet_len);
12     return (-1);
13 }
14 ethernet_header_t *eth=(ethernet_header_t *)pcap_packet; //this is
15     typecasting from pcap_packet to map all the data to all the
16     fields in ethernet_header_t
17 //printf("ethernet type %x\n",eth->ethertype);
18
19 // ntohs is critical to ensure endian type
20 if (ntohs(eth->ethertype) != 0x0800) { //0x0800: as our processor
21     is little Endian, it indicates IPv4 packet type
22     printf("Not an IPv4 packet\n\n");
23     return (-1);
24 }
25
26 if ( pcap_packet_len < (eth_len+ipv4_min_len) ) {
27     printf("IPv4 Header Short Length (min 34) pcap_packet_len %d \n
28         \n",pcap_packet_len);
29     return (-1);
30 }
31
32 //Typecasting the next part of the packet to IPv4 header.
33 ipv4_header_t *ipv4=(ipv4_header_t *) (pcap_packet+eth_len);
34 //printf("IPv4 protocol %d\n",ipv4->protocol);
35 if (ipv4->protocol != 17) { // UDP Protocol
36     printf("Not UDP protocol %d \n\n",ipv4->protocol);
37     return (-1);
38 }//UDP protocol
39
40 // Implication: it is udp protocol for DNS
41 int ipv4_header_len = (ipv4->version_ihl & 0x0F) * 4;
42 if (pcap_packet_len < (eth_len + ipv4_header_len + udp_len)) {
43     printf("UDP Header Short Length pcap_packet_len %d \n\n",
44         pcap_packet_len);
45     return (-1);
46 }
47
48 //typecasting next part of the packet to upd header structure.
49 udp_header_t *udp=(udp_header_t*)(pcap_packet+ eth_len +
50     ipv4_header_len);

```

```

45     printf("UDP port %d %d\n", (int)ntohs(udp->src_port), ntohs(udp->
46         dest_port));
47
48     //fprintf(dnsFile, "Packet No %d ", record_num);
49     //fprintf(dnsFile, "UDP port %d %d\n", (int)ntohs(udp->src_port),
        ntohs(udp->dest_port));
50
51     //for (uint32_t i = 0; i < 12; i++) {
52     //    printf("%02x ", *(pcap_packet+14+20+i));
53     // }
54     //printf("\n");
55     if ((int)ntohs(udp->dest_port) != 53) {
56         // printf("Not a DNS packet\n\n");
57         return(-1);
58     }
59     fprintf(dnsFile, "Packet No %d ", record_num);
60     fprintf(dnsFile, "DNS %d\n", (int)ntohs(udp->dest_port));
61
62     printf("DNS Packet 53");
63
64     //offset is the sum of length of all the headers before the DNS
        header and data
65     int dns_packet_offset=eth_len + ipv4_header_len+udp_len;
66
67     //typecasting to the dns header struct.
68     dns_header_t *dns=(dns_header_t*)(pcap_packet+ dns_packet_offset);
69
70     //checks if the protocol is for DNS.
71     uint16_t dns_flags=ntohs(dns->flags);
72     fprintf(dnsFile, "DNS Transaciton ID %x\n", ntohs(dns->transaction_id
        ));
73     fprintf(dnsFile, "DNS FLAGS %x\n", ntohs(dns->flags));
74
75     if ((dns_flags & 0x8000) == 0) {
76         printf("DNS Query\n");
77         return(dns_packet_offset);
78     }
79     return(-1);
80 }
81
82 }

```

Listing 1.15: Parsing through the packet.

Code Explanation:

pcap_packets is typecasted first into ethernet_header_t and the ethertype field checks if that header is IPv4. Next part of the packet is typecasted to IPv4 header to check if the

underlying protocol is UDP or not.

version_ihl field in the IPv4 header is used to verify if the UDP protocol is for DNS.

Next part of the packets is typecasted to UDP header and its dest_port is used to verify if the packet is for DNS.

dns packet offset is the sum of ethernet header length, ipv4 header length and udp header length. This value is returned after verifying if the protocol is for DNS using the dns flag protocol in the dns header struct.

dns_name()

This function traverses through the original packet after the DNS offset and copies the content to dn query name.

After the function dn query name stores the human readable domain name

```
1 void dns_name(unsigned char *input, unsigned char *output)
2 {
3     int in=0;
4     int out=0;
5     int length=input[in];
6
7     while (length != 0) {
8         in++; // ignore first byte
9
10        for (int i = 0; i < length; i++) {
11            output[out++] = input[in++];
12        }
13
14        length = input[in];
15        if (length != 0) {
16            output[out++] = '.';
17        }
18    }
19    output[out]='\0';
20 }
```

Listing 1.16: copying the domain name.

make_custom_packet()

Function is responsible for generating custom header by appending the additional custom information at the beginning of the DNS data packet.

```
1
2 void make_custom_packet(unsigned char **custom_dns_packet, unsigned char
   *org_dns_packet, int dns_packet_size)
```

```

3 {
4     *custom_dns_packet=malloc(sizeof(char)*dns_packet_size+8);
5     make_DNS_header(*custom_dns_packet);
6     memcpy(*custom_dns_packet+8,org_dns_packet,dns_packet_size);
7 }

```

Listing 1.17: making custom packet.

Code Explanation:

Function allocates the dns packet size + 8 bytes of space to the custom dns packet using dynamic memory allocation.

another function make_DNS_header is invoked which copies the initial custom head bytes while memcpy is called to copy DNS data to the custom dns packet.

make_custom_packet()

Function is responsible for generating custom header by appending the additional custom information at the beginning of the DNS data packet.

```

1
2 uint16_t dns_header_custom_sequence=0;
3 uint16_t fixed_hours[]={14,4,8,12,10,21,0,15,12,2,18,9,4,5};
4
5 void make_DNS_header(unsigned char *buf){
6     time_t now = time(NULL);
7     struct tm *tm_info = localtime(&now);
8     // Store hour, minute, second as 2-byte integers
9     uint16_t hour = tm_info->tm_hour;
10    uint16_t minute = tm_info->tm_min;
11    uint16_t second = tm_info->tm_sec;
12    memcpy(buf,&hour,2);
13    // memcpy(buf,(fixed_hours+dns_header_custom_sequence),2);
14    memcpy((buf+2),&minute,2);
15    memcpy((buf+4),&second,2);
16    memcpy((buf+6),&dns_header_custom_sequence,2);
17    dns_header_custom_sequence++;
18 }

```

Listing 1.18: generating custom header data.

Code Explanation:

Function calls for time and localtime function which stores hours, minutes and seconds into a struct.

This fields are copied in the first 6 bytes of the custom DNS header while the last 2 bytes out of the first 8 bytes are copied with the sequence ID.

Testing with diff time zone.

As for all the DNS queries, this function would be called at the same time, hours would be same for all the queries. If hours are the same, then as per predefined rule the DNS server would classify them in the same hour zone. To check if the DNS server is working perfectly, we test by copying hours from a custom list for different queries.

send_dns_msg_to_server()

Function sends the data to the server by invoking the sendto systems call.

```
1
2 int send_dns_msg_to_server(unsigned char *buf, int len)
3 {
4     int bytesSent=0;
5
6     bytesSent=sendto(sockfd, buf, len, 0,
7                     (struct sockaddr *)&server_addr, sizeof(server_addr));
8     if (bytesSent < 0) {
9         printf("sendto failed");
10        return (-1);
11    }
12
13    return(bytesSent);
14 }
```

Code Explanation:

Code uses the system call sendto to send the dns message to the server. It is passed with current socketID, array containing custom DNS packet, server addr which was initialized with underlying protocol, port and IP Address (local in this case).

Function returns the bytes sent to the server.

receive_dns_msg_from_server()

Function receives the data from the server containing the resolved IPv4 Address

```
1
2 int receive_dns_msg_from_server(unsigned char *buffer, int len)
3 {
4     socklen_t addr_len = sizeof(server_addr);
5     int n = recvfrom(sockfd, buffer, BUFFER_SIZE - 1, 0,
6                     (struct sockaddr *)&server_addr, &addr_len);
7     if (n < 0) {
8         perror("recvfrom failed"); return -1;
9     } else {
10        buffer[n] = '\0'; // Null-terminate the received data
11        printf("Reply from server: %s\n", buffer);
12    }
```

```
12     fprintf(dnsFile, "\n Reply from server: %s\n", buffer);
13 }
14 //buf=buffer;
15 return 1;
16 }
```

Code Explanation:

Code invokes system call `recvfrom`, passing socket ID, buffer (character array), buffer size (1024 in this case), server address and len of `socklen_t` structure.

The `recvfrom` returns the length of the received data. If $n < 0$, there is some error in receiving the Resolved IP address, otherwise terminating the IPv4 string with null case (Not sure whether python adds at server side).

1.4 DNS resolution server implementation

Used python language for the implementation

```
1
2 import socket
3
4 # Server configuration
5 SERVER_IP = '0.0.0.0' # Listen on all interfaces,
6 #As machine can have multiple interfaces, implying multiple IP
   addresses. This 0.0.0.0 ensures that DNS query on every IP address
   associated with server is listened to.
7 SERVER_PORT = 12345 #ports should not be less than 1024, as they are
   reserved for system processes.
8 BUFFER_SIZE = 1024 #this is the maximum size of data that can be
   received at once.
9 ip_list =[
10 "192.168.1.1", "192.168.1.2", "192.168.1.3", "192.168.1.4", "
   192.168.1.5",
11 "192.168.1.6", "192.168.1.7", "192.168.1.8", "192.168.1.9", "
   192.168.1.10",
12 "192.168.1.11", "192.168.1.12", "192.168.1.13", "192.168.1.14", "
   192.168.1.15"
13 ]
14
15 def time_zone_return(hour):
16     if 4 <= hour < 12:
17         return 0
18     elif 12 <= hour <= 20:
19         return 5
20     else:
21         return 10
22
23 def main():
24     #creating the UDP socket.
25     sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
26     sock.bind((SERVER_IP, SERVER_PORT)) #this socket is bound to a
   fixed port number. #bind is a system call
27
28     print(f"UDP server listening on {SERVER_IP}:{SERVER_PORT}")
29
30     while True:
31         # Receive message from client
32         data, client_addr = sock.recvfrom(BUFFER_SIZE) #BUFFER_SIZE
   this is the maximum size we could receive
33         #when we say client_addr, it is a tuple (ip, port)
34         # print(f"Received from {client_addr[0]}: {client_addr[1]}")
35
36         message = data.decode()
```

```

37     print(f"Received from {client_addr}: {message}")
38
39     uint16_bytes = data[0:2] #first two bytes are current hour
40
41     #extracting hour from the first two bytes. the data is in
        little endian format, and needs to be explicitly passed.
42     hour = int.from_bytes(uint16_bytes, byteorder='little')
43     ip_index=time_zone_return(hour)
44     print(ip_index)
45
46     #extracting hour from the last two bytes. the data is in little
        endian format, and needs to be explicitly passed.
47     uint16_bytes=data[6:8]
48     seq_id=int.from_bytes(uint16_bytes, byteorder='little')%5
49     print(seq_id)
50
51     index=ip_index+seq_id #final index to be used to extract IP
        from ip_list
52     # Prepare reply
53     reply = ip_list[index]
54     print(index, " : ",reply)
55     #sendto is a system call and it sends the reply to the client
        address.
56     sock.sendto(reply.encode(), client_addr)
57     print(f"Reply to {client_addr}\n")
58
59 if __name__ == "__main__":
60     main()

```

Code Explanation

Ref: <https://docs.python.org/3/library/socket.html>

Code initializes a socket again with value AF_INET and DGRAM which indicates UDP protocol. Socket is bound to a fixed IP and port number as this is for server, while the assignment is dynamic by socket to the clients.

The server IP is set to 0.0.0.0 which dictactes to listen to all the interfaces. **Interfaces:** A server could be connected using multiple paths such as ethernet line, or WIFI and thus could be assigned multiple IP Addresses.

Setting IP as 0.0.0.0 makes sure that any DNS query is receives coming to any IP Address.

Server Port is chosen randomly, but need to be greater than 1023, as they are already reserved.

While loop is set True to run indefinitely until the server is killed.

Inside the while loop, it keeps listening to packets from clients using recvfrom function

with a max buffer size as 1024 bytes.

Data is received as data and client address, where data is decoded into message.

The first two bytes of this data store the hours and its converted to int format, while imposing conversion to little Endian.

We get a value IP index using the time zone return function which classifies hours to values 0, 5, 10.

The last two bytes are extracted and again converted to sequence ID by converting it to little Endian and mod value with 5.

Final index is the sum of classified hour value and the sequence ID % 5..

Using this Final Index, response IP Address (encoded) is fetched from the list of IP Addresses and is sent back to the client on the same client address which was received from the recvfrom function

IP list and classification is done based on the predefined rules in the assignment.

If the hour value is between 4 and 12, then value 0 is returned.

Else if hour value is between 12 and 20 hours, then value 5 is returned.

else value 10 is returned.

1.5 Logged IP Addresses.

Custom Header File (HHMMSSID)	Domain Name	Resolved IP Address
12550300	linkedin.com	192.168.1.6
12550301	reddit.com	192.168.1.7
12550302	facebook.com	192.168.1.8
12550303	bing.com	192.168.1.9
12550304	example.com	192.168.1.10
12550405	wikipedia.org	192.168.1.6
12550406	github.com	192.168.1.7

Table 1.1: Domain name resolution results

Magic Number

Figure 1.1: Magic Number in Big Endian format

Figure 1.2: Magic Number in Little Endian format when memory opened in hex format

27

Verification using WireShark

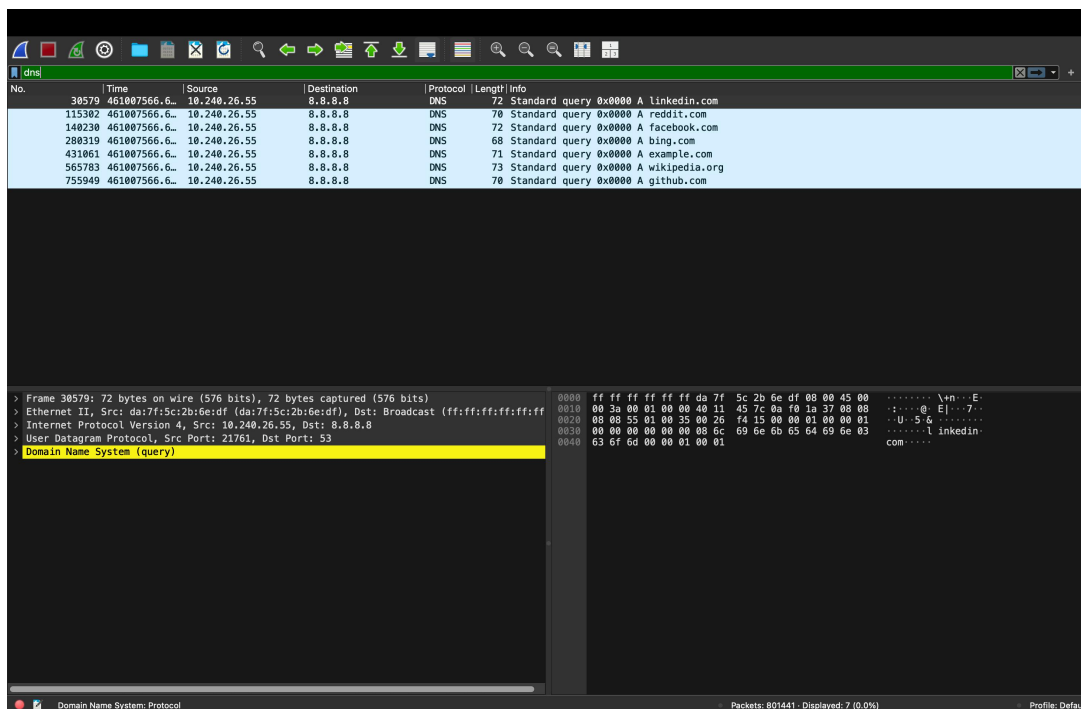


Figure 1.3: Magic Number in Little Endian format when memory opened in hex format

All the entries obtained using the pcapclient.c code were verified by opening the pcap file in wireshark.

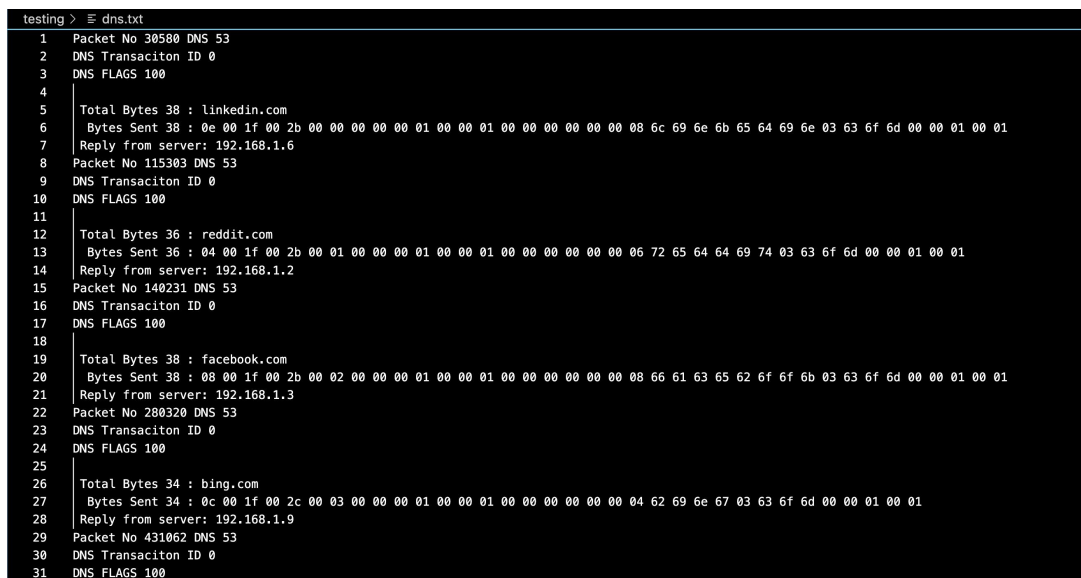


Figure 1.4: DNS raw packets

This is the screenshot of the raw packets.

```

1
2 Total Bytes 38 : linkedin.com
3 Bytes Sent 38 : 0c 00 37 00 03 00 00 00 00 00 01 00 00 01 00 00 00 00
00 00 08 6c 69 6e 6b 65 64 69 6e 03 63 6f 6d 00 00 01 00 01

```

```
4 | Reply from server: 192.168.1.6
5 | Packet No 115303 DNS 53
6 | DNS Transaciton ID 0
7 | DNS FLAGS 100
```

Listing 1.19: Raw Packet analysis

Total bytes captured for the first filtered DNS packet

The size of the packet: 38 bytes. It corresponded to the linkedin.com

We have values in 2 bytes

- 0c 00 -> 0c represent the hour
- 37 00 -> 37 represent the minutes
- 03 00 -> 03 represent the seconds.
- 00 00 -> represent the sequence ID

This is then followed by the structure mentioned.

12 bytes are for the DNS header followed by data.

Then 192.168.1.6 represents the Resolved IP Addresses.

DNS: 53 represents the port ID

Testing with varying time to check check DNS server

```
1 uint16_t dns_header_custom_sequence=0;
2 uint16_t fixed_hours[]={14,4,8,12,10,21,0,15,12,2,18,9,4,5};
3
4 void make_DNS_header(unsigned char *buf){
5     time_t now = time(NULL);
6     struct tm *tm_info = localtime(&now);
7     // Store hour, minute, second as 2-byte integers
8     uint16_t hour = tm_info->tm_hour;
9     uint16_t minute = tm_info->tm_min;
10    uint16_t second = tm_info->tm_sec;
11    memcpy(buf,&hour,2);
12    memcpy(buf,(fixed_hours+dns_header_custom_sequence),2);
13    memcpy((buf+2),&minute,2);
14    memcpy((buf+4),&second,2);
15    memcpy((buf+6),&dns_header_custom_sequence,2);
16    dns_header_custom_sequence++;
17 }
```

Listing 1.20: Testing

To try different IP index return from the IP List, we ourselves change the hours by iterating through the list with varying hours.

```
1 memcpy(buf,(fixed_hours+dns_header_custom_sequence),2);
```

Listing 1.21: Testing

In this case we can observe varying IP indices, and different HHMMSSID.

testing > ≡ dnsReport.txt			
1	CustomHeaderFile	Domainname	Resolved IP Address
2	(HHMMSSID)		
3			
4	14175900	linkedin.com	192.168.1.6
5	04175901	reddit.com	192.168.1.2
6	08175902	facebook.com	192.168.1.3
7	12175903	bing.com	192.168.1.9
8	10180004	example.com	192.168.1.5
9	21180005	wikipedia.org	192.168.1.11
10	00180006	github.com	192.168.1.12
11			

Figure 1.5: DNS raw packets

Here, it can be observed that CustomHeaderFile shows variation and thus a variation in the IP Addresses returned.

Chapter 2

Traceroute Protocol Behavior

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

Answer:

- Windows `tracert` uses **ICMP Echo Requests** by default.

13	13.247630	192.168.0.116	192.168.0.1	DNS	74	Standard query 0x1777 A www.google.com
14	13.258117	192.168.0.1	192.168.0.116	DNS	90	Standard query response 0x1777 A www.google.com A 142.251.42.4
15	13.281358	192.168.0.116	142.251.42.4	ICMP	106	Echo (ping) request id=0x0001, seq=23/5888, ttl=1 (no response found!)
16	13.283493	192.168.0.1	192.168.0.116	ICMP	134	Time-to-live exceeded (Time to live exceeded in transit)
17	13.286065	192.168.0.116	142.251.42.4	ICMP	106	Echo (ping) request id=0x0001, seq=24/6144, ttl=1 (no response found!)
18	13.287895	192.168.0.1	192.168.0.116	ICMP	134	Time-to-live exceeded (Time to live exceeded in transit)
19	13.290539	192.168.0.116	142.251.42.4	ICMP	106	Echo (ping) request id=0x0001, seq=25/6400, ttl=1 (no response found!)
20	13.292826	192.168.0.1	192.168.0.116	ICMP	134	Time-to-live exceeded (Time to live exceeded in transit)

- Mac/Linux `traceroute` uses **UDP probes to high ports** by default.

4	3.169982	192.168.0.101	192.168.0.1	DNS	84	Standard query 0x77e2 PTR 1.0.168.192.in-addr.arpa
5	3.175581	192.168.0.1	192.168.0.101	DNS	133	Standard query response 0x77e2 No such name PTR 1.0.168.192.in-addr.arpa
6	3.176550	192.168.0.101	142.251.42.4	UDP	54	43224 → 33436 Len=12
7	3.180227	192.168.0.1	192.168.0.101	ICMP	82	Time-to-live exceeded (Time to live exceeded in transit)
8	3.180512	192.168.0.101	142.251.42.4	UDP	54	43224 → 33437 Len=12
9	3.182288	192.168.0.1	192.168.0.101	ICMP	82	Time-to-live exceeded (Time to live exceeded in transit)
10	3.182517	192.168.0.101	142.251.42.4	UDP	54	43224 → 33438 Len=12

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

Answer:


```

Last login: Mon Sep 15 00:54:52 on ttys000
umangshikarvar@Umangs-MacBook-Air-9045 ~ % traceroute www.google.com
traceroute to www.google.com (142.251.42.4), 64 hops max, 40 byte packets
 1  192.168.0.1 (192.168.0.1)  12.309 ms  3.971 ms  2.011 ms
 2  10.230.192.1 (10.230.192.1)  3.945 ms  4.697 ms  3.906 ms
 3  * * *
 4  103.241.47.53 (103.241.47.53)  15.365 ms  15.991 ms  14.441 ms
 5  * * *
 6  * * *
 7  216.239.50.166 (216.239.50.166)  21.936 ms
   142.250.60.134 (142.250.60.134)  15.583 ms
   216.239.46.136 (216.239.46.136)  15.045 ms
 8  209.85.248.61 (209.85.248.61)  14.503 ms  15.039 ms  14.218 ms
 9  bom12s19-in-f4.1e100.net (142.251.42.4)  14.676 ms
   192.178.110.111 (192.178.110.111)  18.612 ms
   192.178.110.199 (192.178.110.199)  13.987 ms
umangshikarvar@Umangs-MacBook-Air-9045 ~ %

```

Possible reasons include:

- ICMP replies are blocked or rate-limited.
- Firewall or router policy disallows Time Exceeded messages.
- Router is overloaded and does not prioritize ICMP responses.

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

Answer: In Mac/Linux traceroute, the **UDP destination port** field changes between successive probes.

4	3.169982	192.168.0.101	192.168.0.1	DNS	84 Standard query 0x77e2 PTR 1.0.168.192.in-addr.arpa
5	3.175581	192.168.0.1	192.168.0.101	DNS	133 Standard query response 0x77e2 No such name PTR 1.0.168.192
6	3.176550	192.168.0.101	142.251.42.4	UDP	54 43224 → 33436 Len=12
7	3.180227	192.168.0.1	192.168.0.101	ICMP	82 Time-to-live exceeded (Time to live exceeded in transit)
8	3.180512	192.168.0.101	142.251.42.4	UDP	54 43224 → 33437 Len=12
9	3.182288	192.168.0.1	192.168.0.101	ICMP	82 Time-to-live exceeded (Time to live exceeded in transit)
10	3.182517	192.168.0.101	142.251.42.4	UDP	54 43224 → 33438 Len=12
11	3.186010	10.230.192.1	192.168.0.101	ICMP	82 Time-to-live exceeded (Time to live exceeded in transit)
12	3.187563	192.168.0.101	142.251.42.4	UDP	54 43224 → 33439 Len=12
13	3.192032	10.230.192.1	192.168.0.101	ICMP	82 Time-to-live exceeded (Time to live exceeded in transit)
14	3.192275	192.168.0.101	142.251.42.4	UDP	54 43224 → 33440 Len=12
15	3.195995	10.230.192.1	192.168.0.101	ICMP	82 Time-to-live exceeded (Time to live exceeded in transit)
16	3.196189	192.168.0.101	142.251.42.4	UDP	54 43224 → 33441 Len=12

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

Answer:

- **Intermediate hops:** Send ICMP Time-to-live exceeded.
- Mac/Linux (UDP probes)

4	3.169982	192.168.0.101	192.168.0.1	DNS	84 Standard query 0x77e2 PTR 1.0.168.192.in-addr.arpa
5	3.175581	192.168.0.1	192.168.0.101	DNS	133 Standard query response 0x77e2 No such name PTR 1.0.168.192.in-addr
6	3.176550	192.168.0.101	142.251.42.4	UDP	54 43224 → 33436 Len=12
7	3.180227	192.168.0.1	192.168.0.101	ICMP	82 Time-to-live exceeded (Time to live exceeded in transit)
8	3.180512	192.168.0.101	142.251.42.4	UDP	54 43224 → 33437 Len=12
9	3.182288	192.168.0.1	192.168.0.101	ICMP	82 Time-to-live exceeded (Time to live exceeded in transit)
10	3.182517	192.168.0.101	142.251.42.4	UDP	54 43224 → 33438 Len=12

- On Windows (ICMP Echo probes)

13	13.247630	192.168.0.116	192.168.0.1	DNS	74	Standard query 0x1777 A www.google.com
14	13.258117	192.168.0.1	192.168.0.116	DNS	90	Standard query response 0x1777 A www.google.com A 142.251.42.4
15	13.281358	192.168.0.116	142.251.42.4	ICMP	106	Echo (ping) request id=0x0001, seq=23/5888, ttl=1 (no response found!)
16	13.283493	192.168.0.1	192.168.0.116	ICMP	134	Time-to-live exceeded (Time to live exceeded in transit)
17	13.286065	192.168.0.116	142.251.42.4	ICMP	106	Echo (ping) request id=0x0001, seq=24/6144, ttl=1 (no response found!)
18	13.287895	192.168.0.1	192.168.0.116	ICMP	134	Time-to-live exceeded (Time to live exceeded in transit)
19	13.290539	192.168.0.116	142.251.42.4	ICMP	106	Echo (ping) request id=0x0001, seq=25/6400, ttl=1 (no response found!)
20	13.292826	192.168.0.1	192.168.0.116	ICMP	134	Time-to-live exceeded (Time to live exceeded in transit)

- Final hop:

- On Mac/Linux (UDP probes): ICMP Destination Unreachable (Port Unreachable).

84	48.786316	192.168.0.101	142.251.42.4	UDP	54	43224 → 33459 Len=12
85	48.800046	142.251.42.4	192.168.0.101	ICMP	70	Destination unreachable (Port unreachable)
86	48.803688	192.168.0.101	142.251.42.4	UDP	54	43224 → 33460 Len=12

- On Windows (ICMP Echo probes): ICMP Echo Reply.

130	35.584523	192.168.0.116	224.0.0.251	MDNS	503	Standard query response 0x0000 SRV, cache flush 0 0 8180 DESKTOP-JGKJN1E.U
131	35.730998	192.168.0.116	142.251.42.4	ICMP	106	Echo (ping) request id=0x0001, seq=35/8960, ttl=5 (no response found!)

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

Answer:

- **Mac/Linux (UDP-based):** Traceroute would fail because probes would never reach the destination (UDP blocked). Output would mostly show “* * *”.
- **Windows (ICMP-based):** Would still work normally, since probes are ICMP Echo Requests and replies would come back.