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

2 Traceroute Protocol Behavior **27**

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 preferrable for a user to input human-friendly domain names such as www.google.com. However, to make these interpretable to the networking protcols 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

```
int main(int argc, char *argv[]) {

   // if (argc != 2) {
   //     printf("Usage: %s <pcap file>\n", argv[0]);
   //     return 1;
   // }

    printf("PCAP Client\n");
    //char *filename = argv[1];
    //char *filename = "./p.pcap";

    // Set the pcap file to be processed
    char *filename = "./6.pcap";

    //file which would be used to anayze the DNS packets
    char *dnsFileName="./dns.txt";

    dnsFile = fopen(dnsFileName, "w");

    dnsReportFp = fopen("./dnsReport.txt", "w");


    if (!dnsReportFp) {
        printf("Failed to open file\n");
        return -1;
    }

/*
    Opening the pcap file in the binary format which is stored in the little Endian format in the memory.
*/
    FILE *fp = fopen(filename, "rb");

    if (!fp) {
        printf("Failed to open file\n");
        return -1;
    }
```

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.

```

    create_dns_client_socket(); //we initialize the client socket and configures the server_address

    pcap_hdr_t header; //will store the header of the pcap file

/*
pcap file opened in binary format
fread : C library call to read the binary data.
It copies the first sizeof(header) amount of content to the structure.
This structure is the header of the pcap file
*/

    size_t read_bytes = fread(&header, 1, sizeof(header), fp);
    if (read_bytes != sizeof(header)) {
        printf("Failed to read pcap header\n");
        fclose(fp);
        return -1;
    }

    //printing the header of the pcap file
    print_global_header(&header);

/*

Magic number is used to validate the type of the file.
Magic number : 0xa1b2c3d4 confirms pcap file
*/

    if(header.magic_number != 0xa1b2c3d4){
        printf("%s is not valid pcap file\n",argv[1]);
        exit(-1);
    }
```

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. xa1b2c3d4 is the magic number for the pcap filetype. Code exists in case the magic number obtained does not match the xa1b2c3d4, as it indicates improper file type.

```

fprintf(dnsReportFp,"\tCustomHeaderFile\tDomainname\t\tResolved IP Address\n");
    fprintf(dnsReportFp,"\t  (HHMMSSID)\n\n");

    while(len > 0){
        record_packet1=NULL;
        printf("packet no %d\n",record_num++);

/*

read_pcap_packet is called, which fetches the length of the packet.
Importantly it also allocates memory to the packet read and record_packet1 points to the array where packet is stored.

*/

        len=read_pcap_packet(fp,&record_packet1);


        if(len == 0 ){
            printf("end of file\n"); //len
            break;
        }

        if(len == -1){
            printf("Error in the reading the packet\n");
            printf("Breaking the loop\n");
            break;
        }

        //printf("address %p",(void *)record_packet1);
        //printf("Packet data (first 32 bytes or less):\n");
        //for (uint32_t i = 0; i < 32; i++) {
        //      printf("%02x ", *(record_packet1+i));
        //}
        //printf("\n");
        //ethernet_header_t *eth=(ethernet_header_t *)record_packet1;
        //printf("ethernet type %x\n",eth->ethertype);
        //if (ntohs(eth->ethertype) != 0x0800) {
        //    printf("Not an IPv4 packet\n\n");
        //    return (-1);
        //}

        //getting the offset in the packet to point at the DNS data

        dns_offset=parse_pcap_packet(record_packet1,len);
```

```
        dns_packet_size=len-dns_offset;
        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.

```

if(dns_offset > 0) // Meaning DNS packet
        {
            //printf("DNS packet\n");
            //fprintf(dnsFile,"DNS offset %d %d\n",dns_offset,len);
            //for(int i=0;i< (dns_packet_size);i++)
            //    fprintf(dnsFile,"%02x ", *(record_packet1+i+
                dns_offset));

            //dns_name extracts the human readable domain name by
                accessing the DNS data in the packet.
            dns_name((record_packet1+dns_offset+12),dn_query_name);


            fprintf(dnsFile,"\n Total Bytes %d : %s",
                dns_custom_packet_size,dn_query_name);

            //custom_dns_packet structure stores the final packet
                containing the custom data header along with the DNS
                data.
```

```
            make_custom_packet(&custom_dns_packet,(record_packet1+dns_offset),dns_packet_size);

            // custom_dns_packet is sent to the server to get the ipv4 address.
            int bytesSent=send_dns_msg_to_server(custom_dns_packet,dns_custom_packet_size);
            fprintf(dnsFile,"\n  Bytes Sent %d : ",bytesSent);
            for(int i=0;i< (dns_packet_size+8);i++)
                fprintf(dnsFile,"%02x ", *(custom_dns_packet+i));
            //free(custom_dns_packet);
            //usleep(100000);

            //dns_reply_ip is contains the received IP address from the server.
            receive_dns_msg_from_server(dns_reply_ip,10);

            // custom_dns_packet is type casted to the structure storing the custom header data.
            dns_custom_header_t *dns_ch=(dns_custom_header_t*)(custom_dns_packet);

            //final data is stored to the dnsReport
            fprintf(dnsReportFp,"\t%02d%02d%02d%02d\t\t%s\t\t%s\n",
                            (int)dns_ch->hour,
                            (int)dns_ch->min,
                            (int)dns_ch->sec,
                            (int)dns_ch->seq_no,
                            dn_query_name,
                            dns_reply_ip

                            );
            free(custom_dns_packet);
        }
        //printf("packet no %d\n",record_num++);
        if(record_packet1 != NULL)
            free(record_packet1);
    }
```

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

```c
int sockfd;
struct sockaddr_in server_addr;

FILE *dnsFile;
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.

```c
/*
pcap file structure:
    24 bytes pcap header
    16 bytes record header
    incl_len bytes record or packet (Can be for any type)
    (this repeats)
*/

// 24 Bytes for PCAP Header
typedef struct pcap_hdr_s {

    uint32_t magic_number;   /* magic number */
    uint16_t version_major;  /* major version number */
    uint16_t version_minor;  /* minor version number */
    uint32_t  thiszone;       /* GMT to local correction */
    uint32_t sigfigs;        /* accuracy of timestamps */
    uint32_t snaplen;        /* max length of captured packets, in octets */
    uint32_t network;        /* data link type */

} pcap_hdr_t;
```

Listing 1.6: Pcap file structure and pcap file header

Understanding the pcap file stucture

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 stucture

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

```
/*
record header structure:
    This is a 16byte header before every record in the PCAP file.
    Third entry of this header gives information about the length of
       incl_len
    While forth entry gives the original length of the packet, in case
       cropped
*/

typedef struct pcaprec_hdr_s {
    uint32_t ts_sec;         /* timestamp seconds */
    uint32_t ts_usec;        /* timestamp microseconds */
    uint32_t incl_len;       /* number of octets of packet saved in
       file */
    uint32_t orig_len;       /* actual length of packet */
} 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

```c
typedef struct ethernet_header_s{
    uint8_t dest_mac[6];
    uint8_t src_mac[6];
    uint16_t ethertype;
} 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

```c
// IPv4 header structure
typedef struct ipv4_header_s {
    uint8_t version_ihl;
    uint8_t tos;
    uint16_t total_length;
    uint16_t identification;
    uint16_t flags_frag_offset;
    uint8_t ttl;
    uint8_t protocol; //protocol of the entry
    uint16_t header_checksum;
    uint32_t src_ip;
    uint32_t dest_ip;
} 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

```c
typedef struct udp_header_s {
    uint16_t src_port;
    uint16_t dest_port; //this should be 53 for DNS.
    uint16_t length;
    uint16_t checksum;
} 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

```c
typedef struct dns_header_s {
    uint16_t transaction_id;
    uint16_t flags;
    uint16_t questions;
    uint16_t answer_rrs;
    uint16_t authority_rrs;
    uint16_t additional_rrs;
} 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.

```c
typedef struct dns_custeum_header_s {
    uint16_t hour;
    uint16_t min;
    uint16_t sec;
    uint16_t seq_no;
} 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.

```c
int create_dns_client_socket()
{
    if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
        printf("socket failed");
        return (-1);
    }
    memset(&server_addr, 0, sizeof(server_addr));
    server_addr.sin_family = AF_INET; //this is a standard practice to assign the value '2' for UDP and TCP
    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
    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.
    return(1);

}
```

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.

mmset 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 initalized 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 funtion 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.

```

int read_pcap_packet(FILE *fp,unsigned char **record_packet)
{
    pcaprec_hdr_t record_hdr;
    // Read the next record header

    int len = fread(&record_hdr, 1, sizeof(record_hdr), fp);
    if(len < 1){
        printf("End of file\n");
        return(0);
    }

    if (len != sizeof(record_hdr)) {
        printf("Failed to read record header %d\n",len);
        return (-1);
    }

    // Print the record header fields
    //printf("Timestamp: %u.%06u seconds\n", record_hdr.ts_sec,
       record_hdr.ts_usec);
    //printf("Captured Length: %u bytes\n", record_hdr.incl_len);
    //printf("Original Length: %u bytes\n", record_hdr.orig_len);

/*

incl_len: Field type in the record header struct
Allocation of char array with size being incl_len + 10 bytes extra

*/

    *record_packet=malloc(sizeof(char)*record_hdr.incl_len+10); //
       allocated 10 bytes extra
    if( *record_packet== NULL){
        printf("Memory allocaiton failed for record_packet\n");
        return(-1);
    }

/*

len stores the length of the packet
    if len == 0:
        end of file
    else if len != incl_len:
        error
```

```
    else :
        packet read correctly and size is returned

*/

    len =fread(*record_packet, 1, record_hdr.incl_len, fp);
    if(len < 1){
        printf("End of file\n");
        return(0);
    }
    if (len != record_hdr.incl_len) {
        printf("Failed to read record packet readd %d  expected %d\n",
            len,record_hdr.incl_len);
        return(-1);
    }

    return(record_hdr.incl_len);

}
```

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.

```
int parse_pcap_packet(unsigned char *pcap_packet,int pcap_packet_len)
{
    // Parse Ethernet header
```

```
    //printf("Parsing\n");
    int eth_len=14;
    int ipv4_min_len=20;
    int udp_len=8;

    if (pcap_packet_len < eth_len) {
        printf("Packet Ethernet header short pcap_packet_len %d\n\n",
            pcap_packet_len);
        return (-1);
    }
    ethernet_header_t *eth=(ethernet_header_t *)pcap_packet; //this is
        typecasting from pcap_packet to map all the data to all the
        fields in ethernet_header_t
    //printf("ethernet type %x\n",eth->ethertype);

    // ntohs is critical to ensure endian type
    if (ntohs(eth->ethertype) != 0x0800) { //0x0800: as our processor
        is little Endian, it indicates IPv4 packet type
        printf("Not an IPv4 packet\n\n");
        return (-1);
    }

    if ( pcap_packet_len < (eth_len+ipv4_min_len) ) {
        printf("IPV4 Header Short Length (min 34) pcap_packet_len %d \n
            \n",pcap_packet_len);
        return (-1);
    }

    //Typecasting the next part of the packet to IpV4 header.
    ipv4_header_t *ipv4=(ipv4_header_t *)(pcap_packet+eth_len);
    //printf("IPV4 protocol %d\n",ipv4->protocol);
    if (ipv4->protocol != 17) { // UDP Protocol
        printf("Not UDP protocol %d \n\n",ipv4->protocol);
        return (-1);
    }//UDP protocol

    // Implication: it is udp protocol for DNS
    int ipv4_header_len = (ipv4->version_ihl & 0x0F) * 4;
    if (pcap_packet_len < (eth_len + ipv4_header_len + udp_len)) {
        printf("UDP Header Short Length  pcap_packet_len %d \n\n",
            pcap_packet_len);
        return (-1);

    }

    //typecasting next part of the packet to upd header structure.
    udp_header_t *udp=(udp_header_t*)(pcap_packet+ eth_len +
        ipv4_header_len);
```

```
    printf("UDP port %d %d\n",(int)ntohs(udp->src_port),ntohs(udp->dest_port));

    //fprintf(dnsFile,"Packet No %d ",record_num);
    //fprintf(dnsFile,"UDP port %d %d\n",(int)ntohs(udp->src_port),ntohs(udp->dest_port));

    //for (uint32_t i = 0; i < 12; i++) {
    //    printf("%02x ", *(pcap_packet+14+20+i));
    // }
    //printf("\n");
    if ((int)ntohs(udp->dest_port) != 53) {
           // printf("Not a DNS packet\n\n");
            return(-1);
    }
    fprintf(dnsFile,"Packet No %d ",record_num);
    fprintf(dnsFile,"DNS %d\n",(int)ntohs(udp->dest_port));

    printf("DNS Packet 53");

    //offset is the sum of length of all the headers before the DNS header and data
    int dns_packet_offset=eth_len + ipv4_header_len+udp_len;

    //typecasting to the dns header struct.
    dns_header_t *dns=(dns_header_t*)(pcap_packet+ dns_packet_offset);

    //checks if the protocol is for DNS.
    uint16_t dns_flags=ntohs(dns->flags);
    fprintf(dnsFile,"DNS Transaciton ID %x\n",ntohs(dns->transaction_id));
    fprintf(dnsFile,"DNS FLAGS %x\n",ntohs(dns->flags));

     if ((dns_flags & 0x8000) == 0) {
        printf("DNS Query\n");
        return(dns_packet_offset);
    }
    return(-1);
}

}
```

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 valus is returned after verifying if the procotol 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

```
void dns_name(unsigned char *input, unsigned char *output)
{
    int in=0;
    int out=0;
    int length=input[in];

    while (length != 0) {
         in++;  // ignore first byte

         for (int i = 0; i < length; i++) {
             output[out++] = input[in++];
         }

         length = input[in];
         if (length != 0) {
             output[out++] = '.';
         }
     }
     output[out]='\0';
}
```

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.

```

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

```
{
    *custom_dns_packet=malloc(sizeof(char)*dns_packet_size+8);
    make_DNS_header(*custom_dns_packet);
    memcpy(*custom_dns_packet+8,org_dns_packet,dns_packet_size);
}
```

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.

```

uint16_t dns_header_custom_sequence=0;
uint16_t fixed_hours[]={14,4,8,12,10,21,0,15,12,2,18,9,4,5};

void make_DNS_header(unsigned char *buf){
    time_t now = time(NULL);
    struct tm *tm_info = localtime(&now);
    // Store hour, minute, second as 2-byte integers
    uint16_t hour = tm_info->tm_hour;
    uint16_t minute = tm_info->tm_min;
    uint16_t second = tm_info->tm_sec;
    memcpy(buf,&hour,2);
    // memcpy(buf,(fixed_hours+dns_header_custom_sequence),2);
    memcpy((buf+2),&minute,2);
    memcpy((buf+4),&second,2);
    memcpy((buf+6),&dns_header_custom_sequence,2);
    dns_header_custom_sequence++;
}
```

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.

```c

int send_dns_msg_to_server(unsigned char *buf,int len)
{
    int bytesSent=0;

    bytesSent=sendto(sockfd, buf, len, 0,
                (struct sockaddr *)&server_addr, sizeof(server_addr));
    if (bytesSent < 0) {
        printf("sendto failed");
        return (-1);
    }

    return(bytesSent);
}
```

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

```c

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

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

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

```python

import socket

# Server configuration
SERVER_IP = '0.0.0.0'  # Listen on all interfaces,
#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.
SERVER_PORT = 12345 #ports should not be less than 1024, as they are reserved for system processes.
BUFFER_SIZE = 1024 #this is the maximum size of data that can be received at once.
ip_list =[
"192.168.1.1", "192.168.1.2", "192.168.1.3", "192.168.1.4", "192.168.1.5",
"192.168.1.6", "192.168.1.7", "192.168.1.8", "192.168.1.9", "192.168.1.10",
"192.168.1.11", "192.168.1.12", "192.168.1.13", "192.168.1.14", "192.168.1.15"
]

def time_zone_return(hour):
    if 4 <= hour < 12:
        return 0
    elif 12 <= hour <= 20:
        return 5
    else:
        return 10

def main():
    #creating the UDP socket.
    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    sock.bind((SERVER_IP, SERVER_PORT)) #this socket is bound to a fixed port number. #bind is a system call

    print(f"UDP server listening on {SERVER_IP}:{SERVER_PORT}")

    while True:
        # Receive message from client
        data, client_addr = sock.recvfrom(BUFFER_SIZE) #BUFFER_SIZE this is the maximum size we could receive
        #when we say client_addr, it is a tuple (ip, port)
        # print(f"Received from {client_addr[0]}: {client_addr[1]}")

        message = data.decode()
```

```python
        print(f"Received from {client_addr}: {message}")

        uint16_bytes = data[0:2] #first two bytes are current hour

        #extracting hour from the first two bytes. the data is in little endian format, and needs to be explicitly passed.
        hour = int.from_bytes(uint16_bytes, byteorder='little')
        ip_index=time_zone_return(hour)
        print(ip_index)

        #extracting hour from the last two bytes. the data is in little endian format, and needs to be explicitly passed.
        uint16_bytes=data[6:8]
        seq_id=int.from_bytes(uint16_bytes, byteorder='little')%5
        print(seq_id)

        index=ip_index+seq_id #final index to be used to extract IP from ip_list
        # Prepare reply
        reply = ip_list[index]
        print(index," : ",reply)
        #sendto is a system call and it sends the reply to the client address.
        sock.sendto(reply.encode(), client_addr)
        print(f"Reply to {client_addr}\n")

if __name__ == "__main__":
    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 functioon 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

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-ad
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-ad
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.l
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.