MANIPAL INSTITUTE OF TECHNOLOGY
MANIPAL
A Constituent Institution of Manipal University

Department of Computer Science & Engineering

November, 2021

PACKET SNIFFING AND TCP PACKET FILTERING TOOL

A COMPUTER NETWORKS MINI PROJECT REPORT

Submitted by

SUHAIL MOIDIN and ADITYA

In partial fulfillment for the award of the degree of

BACHELOR OF TECHNOLOGY

IN

COMPUTER SCIENCE AND ENGINEERING

Department of Computer Science & Engineering

BONAFIDE CERTIFICATE

Certified that this project report "PACKET SNIFFING AND TCP PACKET FILTERING TOOL" is the bonafide work of "SUHAIL MOIDIN (190905018) and ADITYA (190905006)" who carried out the mini project work under my supervision.

Dr. Ashalatha Nayak	**Manamohana Krishna**
Professor and Head	**Professor**

Submitted to the Viva voce Examination held on ______________________

EXAMINER 1	**EXAMINER 2**

ABSTRACT

TCP stands for **Transmission Control Protocol** a communications standard that enables application programs and computing devices to exchange messages over a network. It is designed to send packets across the internet and ensure the successful delivery of data and messages over networks.

Packet sniffing is the practice of gathering, collecting, and logging some or all packets that pass through a computer network, regardless of how the packet is addressed.General packet sniffing would collect packets comprising all different transmission protocols incuding UDP,DNS etc.

Packet sniffing **with filter** implies analysing the captured packets, identifying the packets which match the filter,which can be an IP address or any transmission protocol,and finally displaying only those packets .

ACKNOWLEDGEMENT

We would like to thank our Computer Network Professor, Mr Manamohana Krishna and our Parents for their guidance and support.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

A C program implementation of packet sniffer with **TCP filter** ,that analyses the captured packets,identifies TCP protocol,seperates it from the remaining packets and displays the packet header contents and fields on the console.

1.1 Broad Purpose

The purpose of this mini project is to understand how the filtering of packets occurs with the help of C library functions and mainly to understand the structure and contents of a TCP packet which is being filtered out.The filtering process also familiriases concepts of 3-way handshake and retransmission in TCP.

1.2 Basic Aim

Designing a tool for capturing traffic and filtering out TCP packets and its content such as ports,flags, window size and other header fields

2. C Implementation of filtering TCP Packets from Raw File

```
#include <pcap.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

/* default snap length (maximum bytes per packet to capture) */
#define SNAP_LEN 1518

/* ethernet headers are always exactly 14 bytes [1] */
#define SIZE_ETHERNET 14

/* Ethernet addresses are 6 bytes */
#define ETHER_ADDR_LEN     6

/* Ethernet header */
struct sniff_ethernet {
        u_char  ether_dhost[ETHER_ADDR_LEN];    /* destination host address */
        u_char  ether_shost[ETHER_ADDR_LEN];    /* source host address */
        u_short ether_type;                     /* IP? ARP? RARP? etc */
};

/* IP header */
struct sniff_ip {
        u_char  ip_vhl;                 /* version << 4 | header length >> 2 */
        u_char  ip_tos;                 /* type of service */
        u_short ip_len;                 /* total length */
        u_short ip_id;                  /* identification */
        u_short ip_off;                 /* fragment offset field */
        #define IP_RF 0x8000            /* reserved fragment flag */
        #define IP_DF 0x4000            /* don't fragment flag */
        #define IP_MF 0x2000            /* more fragments flag */
        #define IP_OFFMASK 0x1fff       /* mask for fragmenting bits */
        u_char  ip_ttl;                 /* time to live */
        u_char  ip_p;                   /* protocol */
        u_short ip_sum;                 /* checksum */
        struct  in_addr ip_src,ip_dst;  /* source and dest address */
};
#define IP_HL(ip)               (((ip)->ip_vhl) & 0x0f)
#define IP_V(ip)                (((ip)->ip_vhl) >> 4)
```

```
/* TCP header */
typedef u_int tcp_seq;

struct sniff_tcp {
        u_short th_sport;               /* source port */
        u_short th_dport;               /* destination port */
        tcp_seq th_seq;                 /* sequence number */
        tcp_seq th_ack;                 /* acknowledgement number */
        u_char  th_offx2;               /* data offset, rsvd */
#define TH_OFF(th)      (((th)->th_offx2 & 0xf0) >> 4)
        u_char  th_flags;
        #define TH_FIN  0x01
        #define TH_SYN  0x02
        #define TH_RST  0x04
        #define TH_PUSH 0x08

        #define TH_ACK  0x10
        #define TH_URG  0x20
        #define TH_ECE  0x40
        #define TH_CWR  0x80
        #define TH_FLAGS        (TH_FIN|TH_SYN|TH_RST|TH_ACK|TH_URG|TH_ECE|
        TH_CWR|TH_PUSH)
        u_short th_win;                 /* window */
        u_short th_sum;                 /* checksum */
        u_short th_urp;                 /* urgent pointer */
};

void
got_packet(u_char *args, const struct pcap_pkthdr *header, const u_char
*packet);


void
got_packet(u_char *args, const struct pcap_pkthdr *header, const u_char
*packet)
{

         static int count = 1;                   /* packet counter */

         /* declare pointers to packet headers */
         const struct sniff_ethernet *ethernet;  /* The ethernet header [1] */
         const struct sniff_ip *ip;              /* The IP header */
         const struct sniff_tcp *tcp;            /* The TCP header */

         int size_ip;
         int size_tcp;
```

```
printf("\nPacket number %d:\n", count);
count++;


/* define ethernet header */
ethernet = (struct sniff_ethernet*)(packet);

/* define/compute ip header offset */
ip = (struct sniff_ip*)(packet + SIZE_ETHERNET);
size_ip = IP_HL(ip)*4;
if (size_ip < 20) {
	printf("   * Invalid IP header length: %u bytes\n", size_ip);
	return;
}

/* print source and destination IP addresses */
printf("       From: %s\n", inet_ntoa(ip->ip_src));
printf("         To: %s\n", inet_ntoa(ip->ip_dst));

/* determine protocol */
switch(ip->ip_p) {
	case IPPROTO_TCP:
		printf("   Protocol: TCP\n");
		break;
	case IPPROTO_UDP:
		printf("   Protocol: UDP\n");
		return;
	case IPPROTO_ICMP:
		printf("   Protocol: ICMP\n");
		return;
	case IPPROTO_IP:
		printf("   Protocol: IP\n");
		return;
	default:
		printf("   Protocol: unknown\n");
		return;
}


//only when packet is TCP,it reaches this code segment ,else returns back
//to loop

/* define/compute tcp header offset */
tcp = (struct sniff_tcp*)(packet + SIZE_ETHERNET + size_ip);
size_tcp = TH_OFF(tcp)*4;
if (size_tcp < 20) {
	printf("   * Invalid TCP header length: %u bytes\n", size_tcp);
	return;
}
```

```
        printf("   Src port:           %d\n", ntohs(tcp->th_sport));
        printf("   Dst port:           %d\n", ntohs(tcp->th_dport));
        printf("   TCP sequence number:%u\n",(u_int)ntohl(tcp->th_seq));
        printf("   TCP ack number:     %u\n",(u_int)ntohl(tcp->th_ack));
        if (tcp->th_flags & TH_ACK)
                puts ("   ACK set");
        if (tcp->th_flags & TH_PUSH)
                puts ("   PUSH set");
        if (tcp->th_flags & TH_SYN)
                puts ("   SYN set");
        if (tcp->th_flags & TH_FIN)
                puts ("   FIN set");
        if (tcp->th_flags & TH_URG)
                puts ("   URG set");
        if (tcp->th_flags & TH_RST)
                puts ("   RST set");
        printf("   window size:        %d\n",ntohs(tcp->th_win));


return;
}

int main()
{


        char errbuf[PCAP_ERRBUF_SIZE];             /* error buffer */
        pcap_t *handle;                            /* packet capture handle */

        char filter_exp[] = "tcp";                 /* filter expression [3] */
        struct bpf_program fp;                     /* compiled filter program
                                                   (expression) */
        bpf_u_int32 mask;                          /* subnet mask */
        bpf_u_int32 net;                  /* ip */
        int num_packets = 10;                      /* number of packets to capture */



        handle = pcap_open_offline("capture.pcapng", errbuf);
        //handle = pcap_open_offline("ftp1.pcapng", errbuf);
        if (handle == NULL) {
                fprintf(stderr, "Couldn't open device , %s\n",  errbuf);
                exit(EXIT_FAILURE);
        }



        /* compile the filter expression */
        if (pcap_compile(handle, &fp, filter_exp, 0, net) == -1) {
                fprintf(stderr, "Couldn't parse filter %s: %s\n",
```

```
                filter_exp, pcap_geterr(handle));
            exit(EXIT_FAILURE);
        }

        /* apply the compiled filter */
        if (pcap_setfilter(handle, &fp) == -1) {
            fprintf(stderr, "Couldn't install filter %s: %s\n",
                filter_exp, pcap_geterr(handle));
            exit(EXIT_FAILURE);
        }

        /* now we can set our callback function */
        pcap_loop(handle, num_packets, got_packet, NULL);

        /* cleanup */
        pcap_freecode(&fp);
        pcap_close(handle);

        printf("\nCapture complete.\n");

return 0;
}
```

Testing/ Output:

The above program is tested using **‘pcap’** files from wireshark as input.
Initially, it is tested with a wireshark capture ‘capture.pcapng’ which is a capture of all the packet traffic ,in that particular network over a short period of time.
Next it is tested with ‘ftp1.pcpang’which is again a capture of all packet traffic activities in that network, but this time , packet traffic involves FTP too which uses TCP to tranfer files.

capture.pcapng:

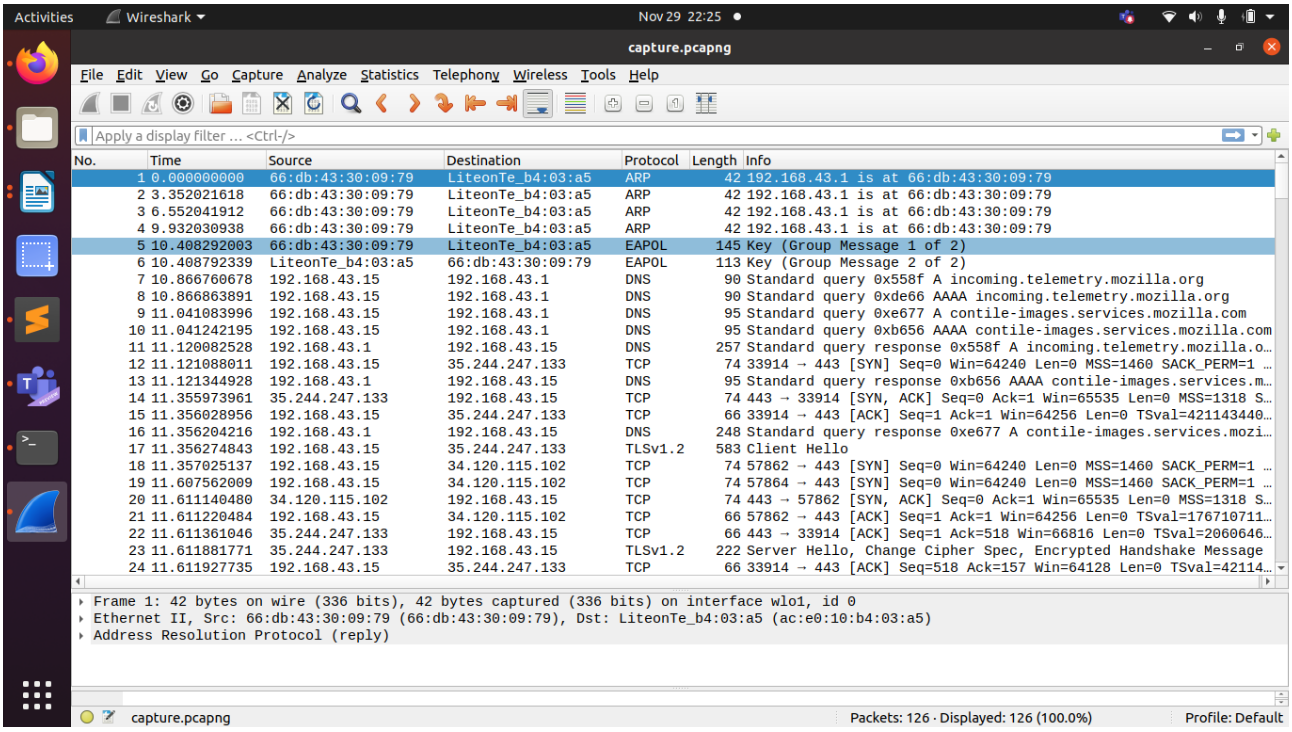

figure 3.1.1

working:

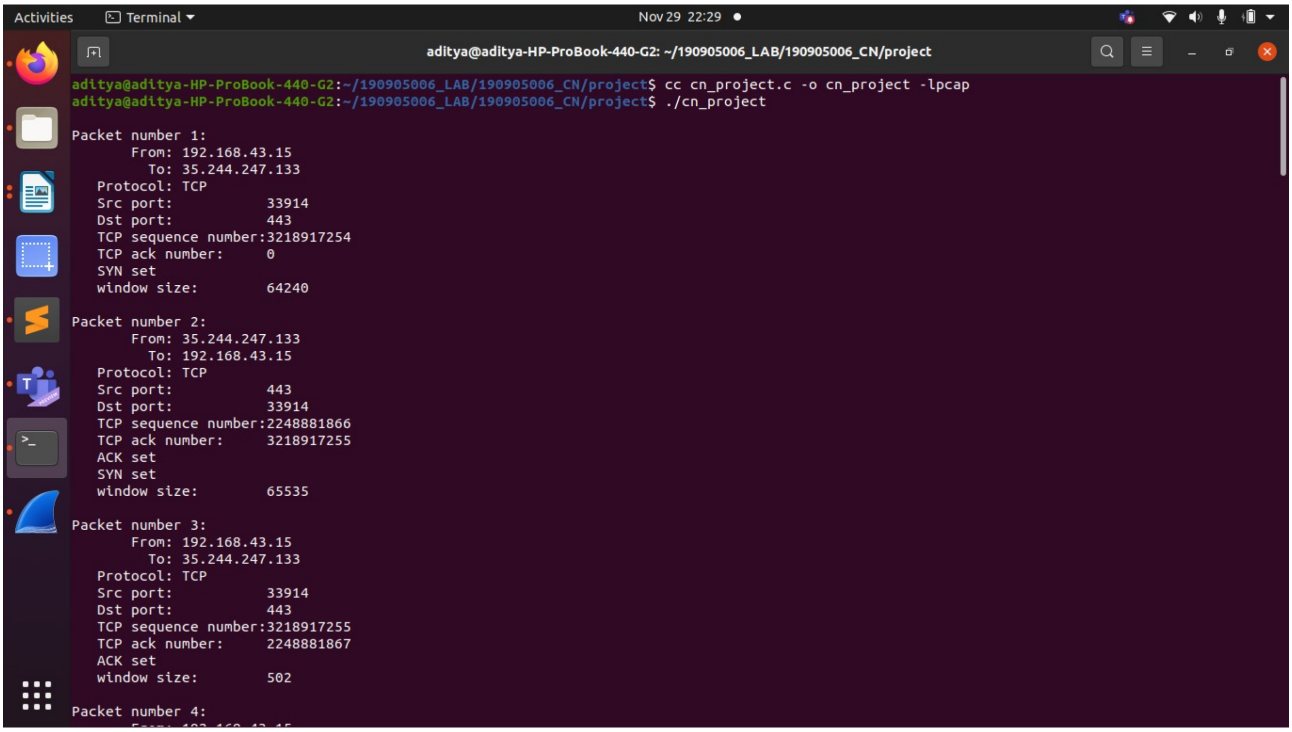

figure 3.1.2

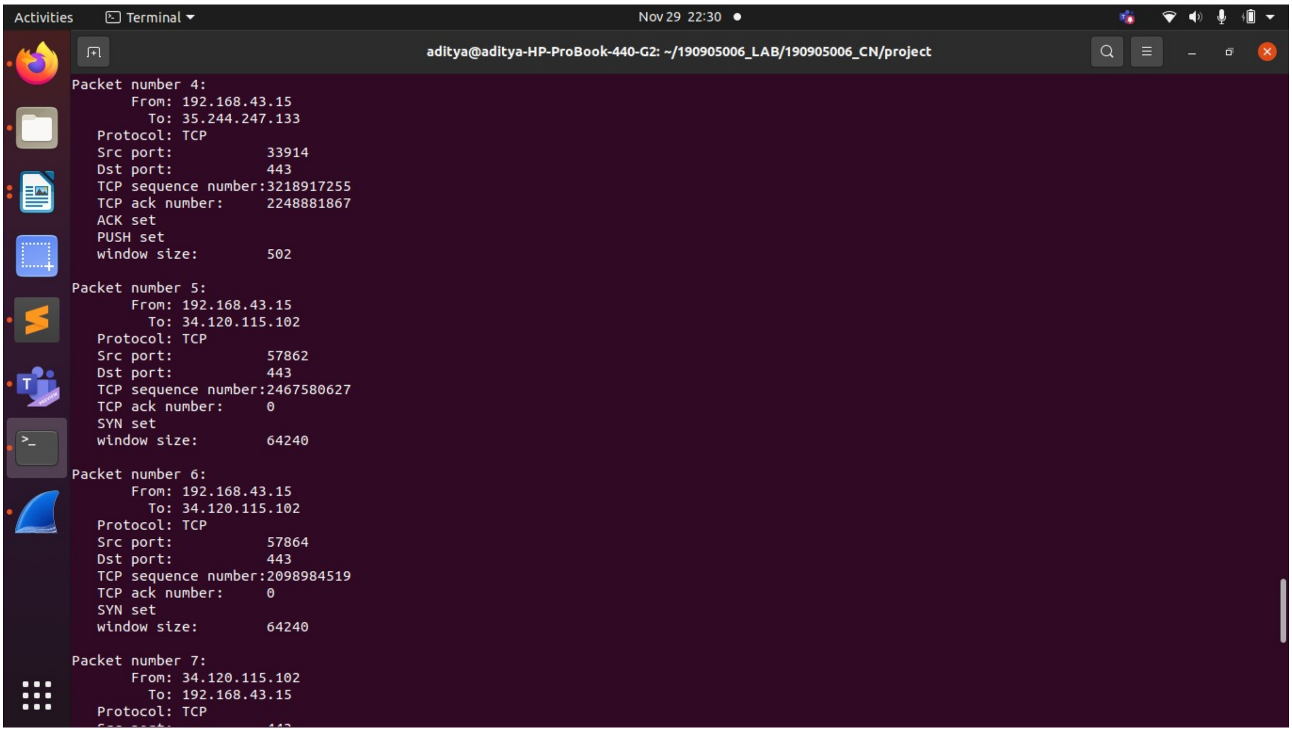

figure 3.1.3

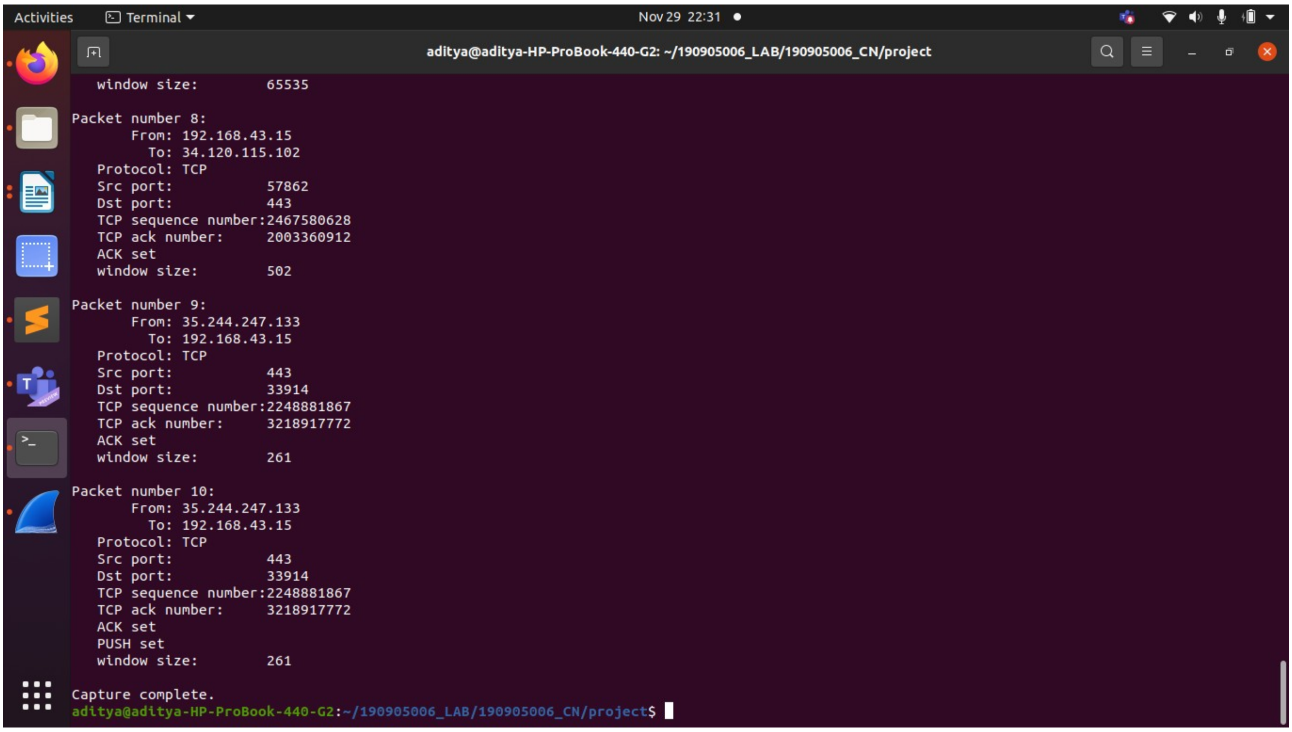

figure 3.1.4

using ftp1.pcapng, to monitor and observe tcp protocol in ftp file transfer:

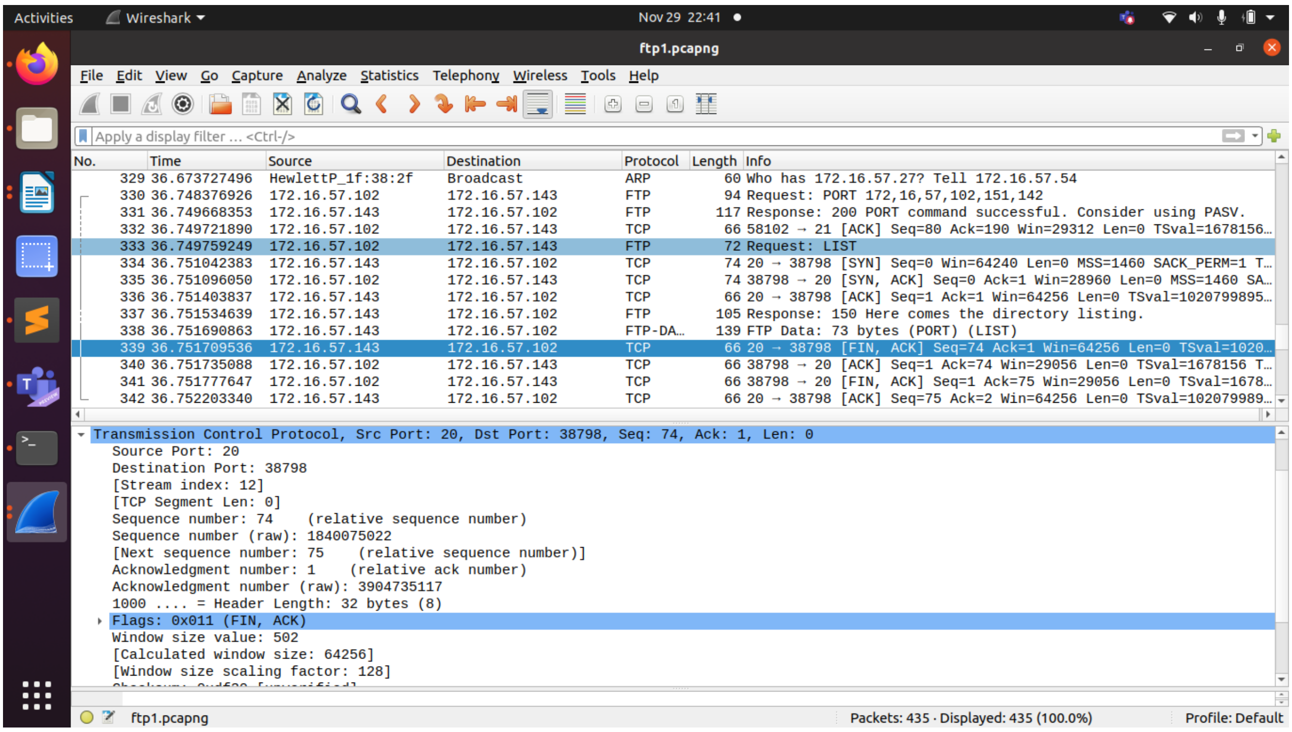

figure 3.2.1

since in the wireshark capture , **file transfer** is seen to be after 300+ packets already captured, the code is accordingly modified to filter packets after skipping unwanted packets.

Capturing from the 50th TCP packet inorder to capture FTP traffic from our input:

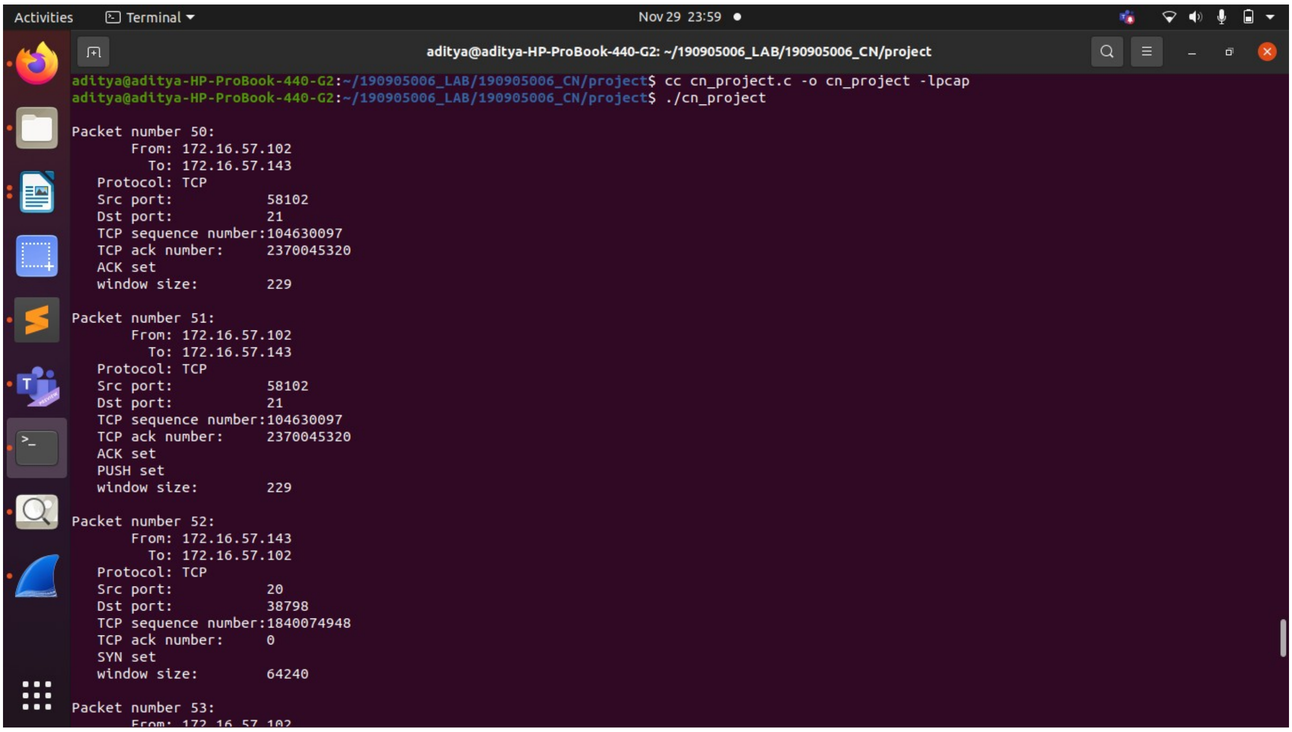

figure 3.2.2

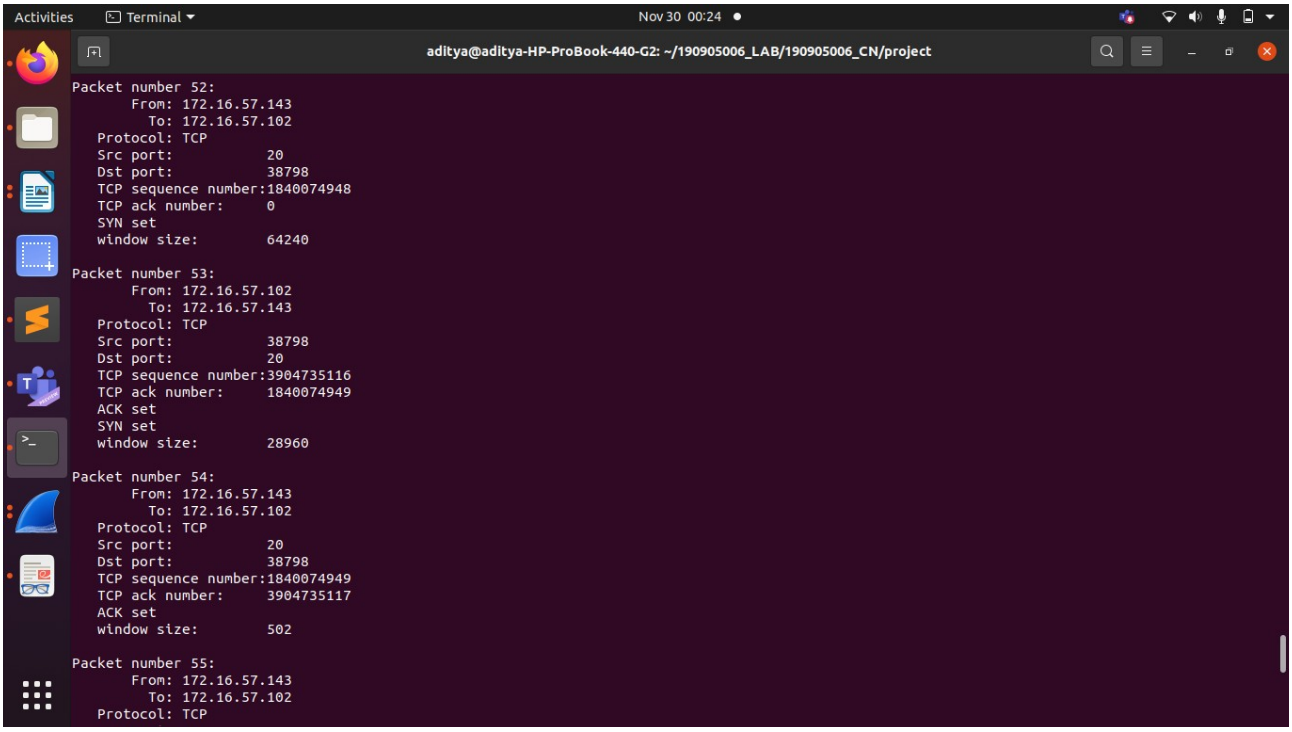

figure 3.2.3

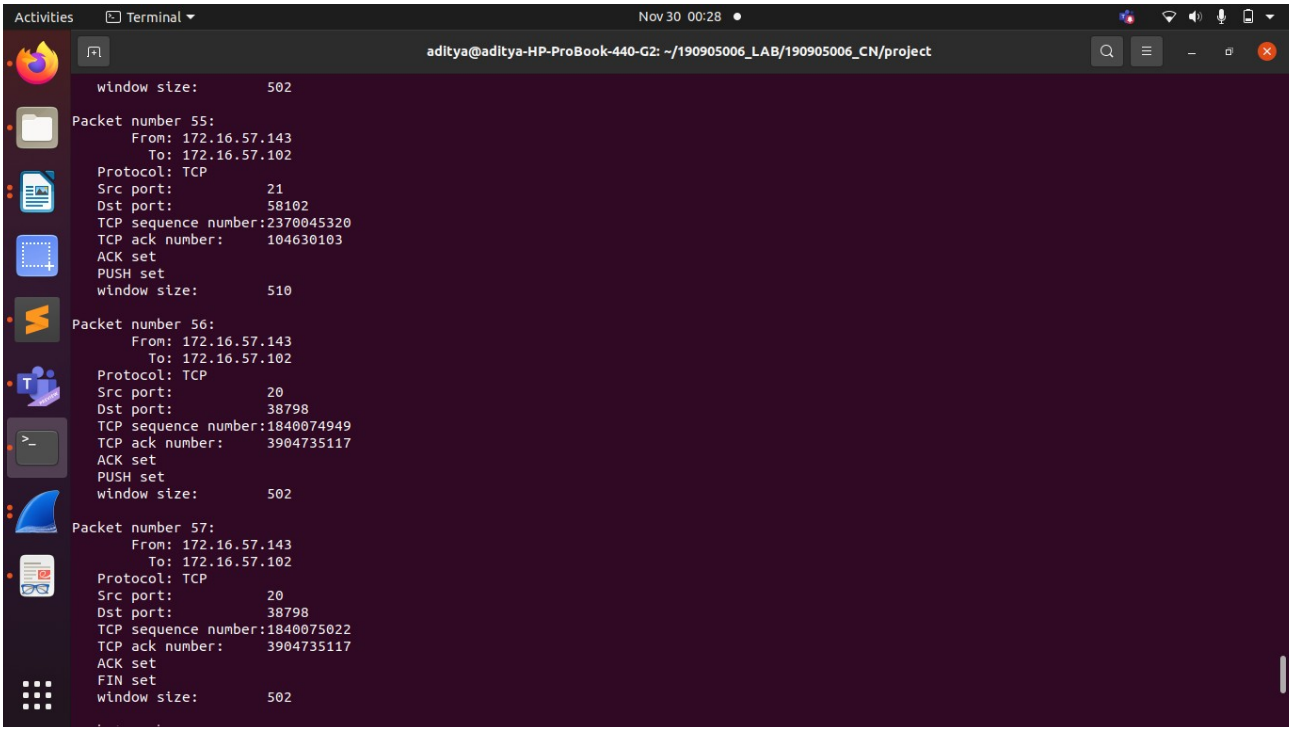

figure 3.2.4

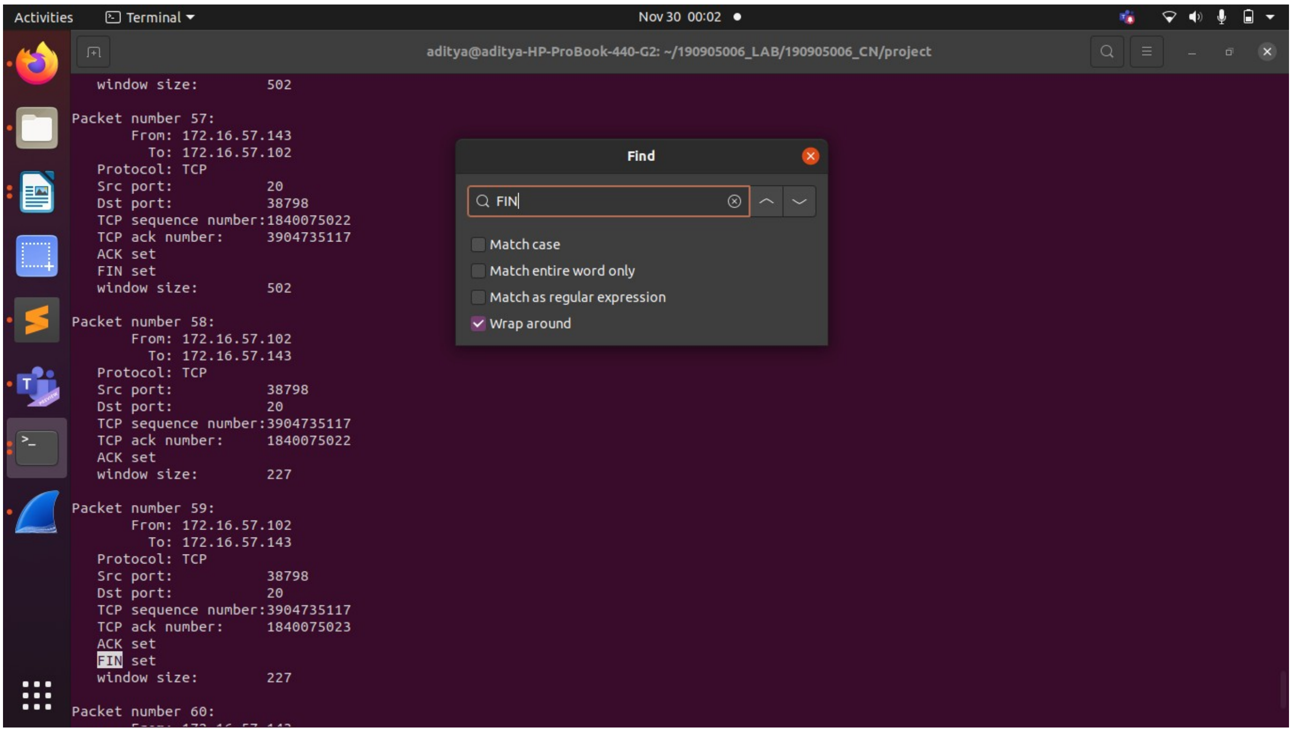

figure 3.2.5

4. ANALYSIS /EXPLANATION

4.1 The Format of a pcap Application:

Here is the general layout of the pcap sniffer and the flow of code:

1. We begin by determining which interface we want to sniff on. In Linux this may be something like eth0. We can either define this device in a string or we can ask pcap to provide us with the name of an interface that will do the job.

2. Initialize pcap. This is where we actually tell pcap what device we are sniffing on. We could sniff on multiple devices but we differentiate between them using file handles. Just like opening a file for reading or writing, we must name our sniffing "session" so we can tell it apart from other such sessions.

3.To ensure that we sniff on specific packets only like only TCP/IP packets, we must create a rule set, "compile" it, and then apply it. This is a three phase process, all of which is closely related. The rule set is kept in a string, and is converted into a format that pcap can read (hence compiling it.) The compilation is done by calling a function within our program. Then we tell pcap to apply it to whichever session we wish for it to filter.

4. Finally, we tell pcap to enter it's primary execution loop. In this state, pcap waits until it has received however many packets we want it to. Every time it gets a new packet in, it calls another function that we have already defined. The function that it calls can do anything we want; it can dissect the packet and print it to the user, it can save it in a file, or it can do nothing at all.

5. After the sniffing needs are satisfied, we close the session and complete.

4.2 Opening the Device for Sniffing:

For the task of sniffing a session, we use pcap_open_offline(). The prototype of this function (from the pcap man page) is as follows:

```
pcap_t *pcap_open_offline(const char *fname, char *errbuf);
```

fname specifies the name of the file to open. The file can have the pcapng file format, although not all pcapng files can be read.

In the code, we use the following code snippet:

```
#include <pcap.h>
  ...
handle = pcap_open_offline("capture.pcapng", errbuf);
if (handle == NULL) {
        fprintf(stderr, "Couldn't open device , %s\n",  errbuf);
        exit(EXIT_FAILURE);
}
```

This code fragment opens the capture.pcapng file, tells it to read however many bytes are present. We are telling it to put the device into promiscuous mode, to sniff until an error occurs, and if there is an error, store it in the string errbuf; it uses that string to print an error message.

4.3 Filtering Traffic:

To filter out the TCP packets from the rest, we use pcap_compile() and pcap_setfilter() function. After we have already called pcap_open_offline() and have a working sniffing session, we can apply our filter. We don’t use our own if/else statements due to the fact that pcap's filter is far more efficient because it does it directly with the BPF filter. Hence, we eliminate numerous steps by having the BPF driver do it directly.
Before applying our filter, we must "compile" it. To compile the program we call pcap_compile(). The prototype defines it as:

```
int pcap_compile(pcap_t *p, struct bpf_program *fp, char *str, int optimize,
    bpf_u_int32 netmask)
```

The first argument is our session handle (pcap_t *handle in our previous example). Following that is a reference to the place we will store the compiled version of our filter. Then comes the expression itself, in regular string format. Next is an integer that decides if the expression should be "optimized" or not (0 is false, 1 is true. Standard stuff.) Finally, we must specify the network mask of the network the filter applies to. The function returns -1 on failure; all other values imply success.
After the expression has been compiled, it is time to apply it. To apply, we use pcap_setfilter(). The prototype defines it as:

```
int pcap_setfilter(pcap_t *p, struct bpf_program *fp)
```

The first argument is our session handle, the second is a reference to the compiled version of the expression.

4.4 The Actual Sniffing:

There are two main techniques for capturing packets. We can either capture a single packet at a time, or we can enter a loop that waits for *n* number of packets to be sniffed before being done. Capturing a single packet can be done by using pcap_next() and capturing via loop can be done using pcap_loop().
The first argument is our session handle. Following that is an integer that tells pcap_loop() how many packets it should sniff for before returning (a negative value means it should sniff until an error occurs). The third argument is the name of the callback function (just its identifier, no parentheses). The last argument is useful in some applications, but many times is simply set as NULL.
Before we can provide an example of using pcap_loop(), we must examine the format of our callback function. We use this format as the prototype for our callback function:

The prototype for pcap_loop() is below:

```
int pcap_loop(pcap_t *p, int cnt, pcap_handler callback, u_char *user)
```

Example:

```
int pcap_loop(pcap_t *p, int cnt, pcap_handler callback, u_char *user)
#include <pcap.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
      char errbuf[PCAP_ERRBUF_SIZE];              /* error buffer */
pcap_t *handle;                               /* packet capture handle */

char filter_exp[] = "tcp";                    /* filter expression [3] */
struct bpf_program fp;                        /* compiled filter program
(expression) */
bpf_u_int32 mask;                             /* subnet mask */
bpf_u_int32 net;                     /* ip */
int num_packets = 20;                         /* number of packets to capture */

 /* Open the session in promiscuous mode */
     handle = pcap_open_offline("capture.pcapng", errbuf);
if (handle == NULL) {
        fprintf(stderr, "Couldn't open device , %s\n",  errbuf);
        exit(EXIT_FAILURE);
}


      }

  /* compile the filter expression */
if (pcap_compile(handle, &fp, filter_exp, 0, net) == -1) {
        fprintf(stderr, "Couldn't parse filter %s: %s\n",
            filter_exp, pcap_geterr(handle));
        exit(EXIT_FAILURE);
}

/* apply the compiled filter */
if (pcap_setfilter(handle, &fp) == -1) {
        fprintf(stderr, "Couldn't install filter %s: %s\n",
            filter_exp, pcap_geterr(handle));
        exit(EXIT_FAILURE);
}

      /* Grab packets via loop */
      pcap_loop(handle, num_packets, got_packet, NULL);
}
```

One cannot arbitrarily define our callback's prototype; otherwise, pcap_loop() would not know how to use the function. So we use this format as the prototype for our callback function:

```
void got_packet(u_char *args, const struct pcap_pkthdr *header,const u_char *packet);
```

It can be noticed that this function has a void return type. This is logical because pcap_loop() wouldn't know how to handle a return value anyway. The first argument corresponds to the last argument of pcap_loop(). Value passed as the last argument to pcap_loop() is passed to the first argument of our callback function every time the function is called. The second argument is the pcap header, which contains information about when the packet was sniffed, how large it is, etc.

The pcap_pkthdr structure is defined in pcap.h as:

```
struct pcap_pkthdr {
        struct timeval ts; /* time stamp */
        bpf_u_int32 caplen; /* length of portion present */
        bpf_u_int32 len; /* length this packet (off wire) */
};
```

The last argument is a pointer to a u_char, and it points to the first byte of a chunk of data containing the entire packet, as sniffed by pcap_loop().

A packet contains many attributes, so it is not really a string, but actually a collection of structures (for instance, a TCP/IP packet would have an Ethernet header, an IP header, a TCP header, and lastly, the packet's payload). This u_char pointer points to the serialized version of these structures. To make any use of it, typecasting is done.

To typecast, one must have the actual structures defined before itself. The following are the structure definitions that is used to describe a TCP/IP packet over Ethernet.

```
/* Ethernet addresses are 6 bytes */
#define ETHER_ADDR_LEN  6

    /* Ethernet header */
    struct sniff_ethernet {
        u_char ether_dhost[ETHER_ADDR_LEN]; /* Destination host address */
        u_char ether_shost[ETHER_ADDR_LEN]; /* Source host address */
        u_short ether_type; /* IP? ARP? RARP? etc */
    };

    /* IP header */
    struct sniff_ip {
        u_char ip_vhl;          /* version << 4 | header length >> 2 */
        u_char ip_tos;          /* type of service */
        u_short ip_len;         /* total length */
        u_short ip_id;          /* identification */
        u_short ip_off;         /* fragment offset field */
    #define IP_RF 0x8000            /* reserved fragment flag */
    #define IP_DF 0x4000            /* don't fragment flag */
    #define IP_MF 0x2000            /* more fragments flag */
    #define IP_OFFMASK 0x1fff       /* mask for fragmenting bits */
        u_char ip_ttl;          /* time to live */
        u_char ip_p;            /* protocol */
        u_short ip_sum;         /* checksum */
        struct in_addr ip_src,ip_dst; /* source and dest address */
```

```
};
#define IP_HL(ip)               (((ip)->ip_vhl) & 0x0f)
#define IP_V(ip)                (((ip)->ip_vhl) >> 4)

/* TCP header */
typedef u_int tcp_seq;

struct sniff_tcp {
        u_short th_sport;       /* source port */
        u_short th_dport;       /* destination port */
        tcp_seq th_seq;         /* sequence number */
        tcp_seq th_ack;         /* acknowledgement number */
        u_char th_offx2;        /* data offset, rsvd */
#define TH_OFF(th)      (((th)->th_offx2 & 0xf0) >> 4)
        u_char th_flags;
#define TH_FIN 0x01
#define TH_SYN 0x02
#define TH_RST 0x04
#define TH_PUSH 0x08
#define TH_ACK 0x10
#define TH_URG 0x20
#define TH_ECE 0x40
#define TH_CWR 0x80
#define TH_FLAGS (TH_FIN|TH_SYN|TH_RST|TH_ACK|TH_URG|TH_ECE|
TH_CWR)
        u_short th_win;         /* window */
        u_short th_sum;         /* checksum */
        u_short th_urp;         /* urgent pointer */
};
```

Since we’re dealing with TCP/IP., we begin by defining the variables and compile-time definitions Then we need to deconstruct the data.

```
/* ethernet headers are always exactly 14 bytes */
#define SIZE_ETHERNET 14

const struct sniff_ethernet *ethernet; /* The ethernet header */
const struct sniff_ip *ip; /* The IP header */
const struct sniff_tcp *tcp; /* The TCP header */
const char *payload; /* Packet payload */

u_int size_ip;
u_int size_tcp;
ethernet = (struct sniff_ethernet*)(packet);
ip = (struct sniff_ip*)(packet + SIZE_ETHERNET);
size_ip = IP_HL(ip)*4;
if (size_ip < 20) {
        printf("   * Invalid IP header length: %u bytes\n", size_ip);
        return;
```

```
}
tcp = (struct sniff_tcp*)(packet + SIZE_ETHERNET + size_ip);
size_tcp = TH_OFF(tcp)*4;
if (size_tcp < 20) {
	printf("   * Invalid TCP header length: %u bytes\n", size_tcp);
	return;
}
```

The u_char pointer is really a variable containing an address in memory.
The address of the structure after the header is the address of that header plus the length of that header. The IP header, unlike the Ethernet header, does not have a fixed length; its length is given, as a count of 4-byte words, by the header length field of the IP header. As it's a count of 4-byte words, it must be multiplied by 4 to give the size in bytes. The minimum length of that header is 20 bytes.
The TCP header also has a variable length. It’s length is given as a number of 4-byte words by the "data offset" field of the TCP header and it’s minimum length is also 20 bytes.

Variable	Location (in bytes)
sniff_ethernet	X
sniff_ip	X + SIZE_ETHERNET
sniff_tcp	X + SIZE_ETHERNET + {IP header length}

Table 4.4.1

The sniff_ethernet structure, being the first in line, is simply at location X. sniff_ip, who follows directly after sniff_ethernet, is at the location X, plus however much space the Ethernet header consumes (14 bytes, or SIZE_ETHERNET). sniff_tcp is after both sniff_ip and sniff_ethernet, so it is location at X plus the sizes of the Ethernet and IP headers (14 bytes, and 4 times the IP header length, respectively).

5. Understanding

From Test 1, TCP packets are succesfuly filtered from general network traffic in a network.
The different fields in a TCP packet header such as source port, destination port, window size,sequence number, acknowledgement number have been identified and printed, (succesful implementation of ntohs(), and ntohl() functions).
It is important to note that the source and destination address are obtained from IP header which helps in connecting hosts, while TCP or transport layer protocols connect process to process.
The flags are also analysed ,with different packets having different flags set.

From Test 2, the behaviour of FTP is understood, which uses TCP to transfer files.
Figure 3.2.3 to Figure 3.2.5 show the TCP packets filtered out during file tranfer.
It is intresting to see the use of 2 ports, 21 for control connection, and 20 for data connection as recalled from theory classes.
Also flags set have been observed. Initially for packet 52, figure 3.2.3 SYN is set, for which ACK is sent back from other host along with its SYN request.File transfer is now started. When file transfer is finished, immediately FIN is seen in data connection indicating FTP data connection is non persistent.

Overall, an understanding is developed on how data packets are captured and how the filters are applied and compiled, with the help of functions from C library ‘libpcap’.Mainly, the STRUCTURE of a TCP SEGMENT has been familiriased with.The significance of various header fields has been noted and for what purpose they have been set,has beenobserved.

6. References

- James F. Kurose & Keith W. Ross, *Computer Networking A Top-Down Approach*, (6e), Pearson Education,2013
- https://www.tcpdump.org/pcap.html
- https://stackoverflow.com/questions/65238453/how-to-find-tcp-retransmissions-while-sniffing-packets-in-c
- https://stackoverflow.com/questions/2072046/how-to-print-flags-in-tcp-header-of-raw-packets-using-libpcap
- https://www.opensourceforu.com/2011/02/capturing-packets-c-program-libpcap/