Network Statistics such as Throughput, Transmission Speed, Average RTT.

A MINI PROJECT REPORT

Submitted by

Ayush Goyal - 190905522

Manan Dahiya - 190905236

In partial fulfilment for the award of the degree of

Bachelor's in technology (B. Tech)

IN

Computer Science & Engineering

MANIPAL INSTITUTE OF TECHNOLOGY
MANIPAL
A Constituent Institution of Manipal University

Department of Computer Science & Engineering

MAY 2023

Department of Computer Science & Engineering

BONAFIDE CERTIFICATE

Certified that this project report "Network Statistics such as Throughput, Transmission Speed, Average RTT." is the bonafide work of:

Ayush Goyal (190905522)

&

Manan Dahiya (190905236)

who carried out the mini project work under my supervision.

Dr. Ashalatha Nayak	**Dr. Krishnamoorthi M**
Head of Department, CSE	**Professor, CSE**

Submitted to the Viva voce Examination held on

____

EXAMINER 1	**EXAMINER 2**

Abstract for the project: “Network Statistics such as Throughput, Transmission Speed, Average RTT”

For our mini project, we are sniffing data packets passed to us in a .pcap file and calculating different types of statistics regarding each packet. At the end, we analyse the overall statistics of the network such as average speed, average packet size, average packet rate and average RTT. For each packet we try to print the host ip, destination ip, source port address, destination port address, packet capture length, packet total length, sequence number and acknowledgement number. For TCP oriented data packets, we have even printed the payload. The whole code is written in C in which we use the libcap library.

The libpcap library was written as part of a more extensive program called TCPDump. The libpcap library allowed developers to write code to receive link-layer packets (Layer 2 in the OSI model) on different flavours of UNIX operating systems without having to worry about the idiosyncrasy of different operating systems' network cards and drivers. Essentially, the libpcap library grabs packets directly from the network cards, which allowed developers to write programs to decode, display, or log the packets.

In our program, we allow the user to input any .pacp file for which we output its statistics. We also allow the user to put a limit to the number of packets to be sniffed.

ACKNOWLEDGEMENT

We would like to thank our professor, Dr. Krishanamoorthi Makkithaya for providing us with the motivation and the knowledge required to implement this project. He was very helpful and got through to us very clearly regarding all topics on the subject. Once the basics were clear, it was an easy feat to have implemented the project.

We would also like to appreciate the infrastructure provided to us by our educational institute, Manipal Institute of Technology, without which we would have a lot of trouble getting familiar with the required software.

We would also like to acknowledge the help of all our peers and lab staff for helping us understand the problem statement, hence enabling us to be able to write code for the same.

We wish to thank our well-wishers and guardians for their undivided support and interests in out well-being and being a constant source of motivation and inspiration.

TABLE OF CONTENTS

Chapter 1: INTRODUCTION TO TCP

1.1 Introduction to Computer Networks

A computer network is defined as an interconnection of multiple devices, known as hosts, connected using multiple paths to send/receive data or media. Computer networks can include numerous devices/ mediums that help communicate between two devices, and these devices/ mediums are known as Network Devices and include routers, switches, hubs, and bridges.

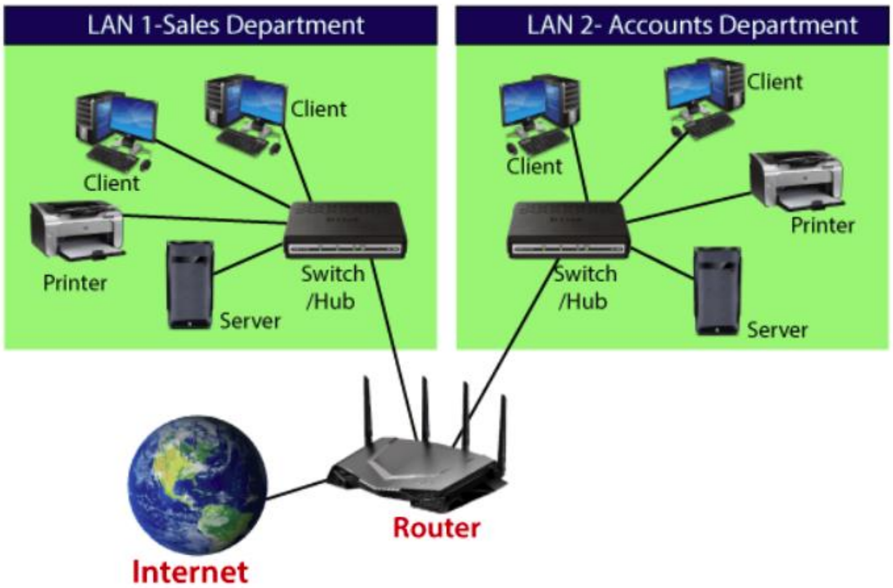


Fig 1.1 A computer network with hosts, switches/ hubs, servers, and routers

1.2 Transport Layer

The transport layer is a central piece of the layered network architecture between the application and network layers. It has the critical role of providing communication services directly to the application processes running on different hosts. The Internet has two protocols—TCP and UDP.

UDP (User Datagram Protocol) is a connectionless protocol, and TCP is a connection-oriented protocol. Usually, UDP (besteffort protocol) is chosen when the speed is more important than the reliability. It's primarily used in video conferencing, live streaming, gaming, etc. It doesn't have mechanisms to handle flow/ congestion/ loss in the network. Hence where reliability is of importance, TCP is preferred over UDP.

1.3 Transmission Control Protocol (TCP)

TCP is a connection-oriented protocol. It is called a connectionoriented protocol because before sending data between two hosts, a connection has to be set up between sender and receiver with three-way hand-shaking.

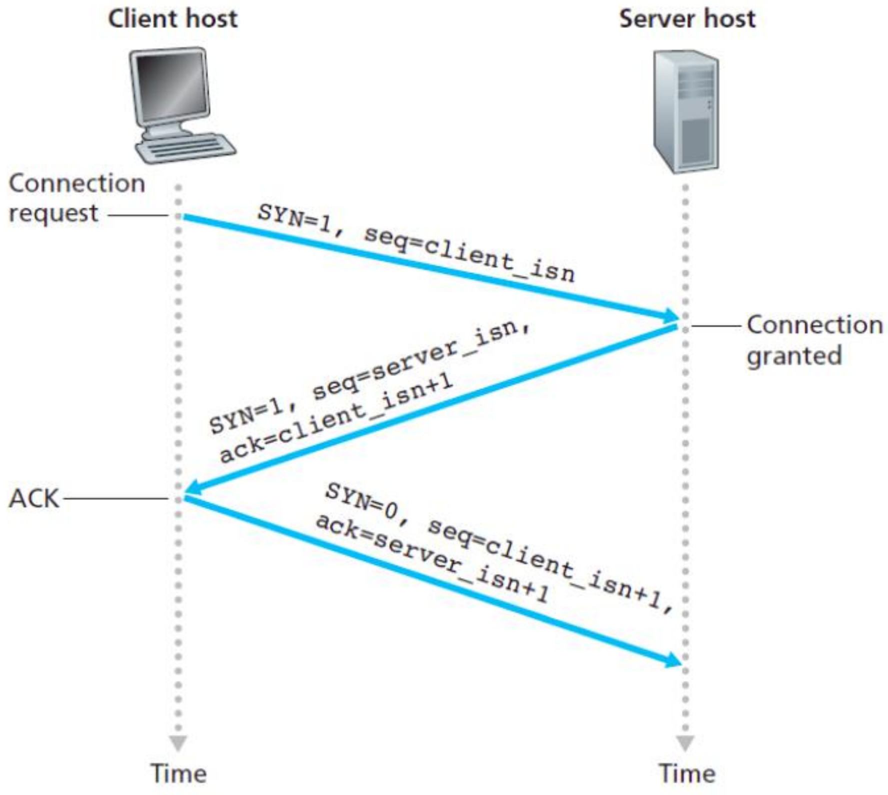


Figure 1.3 Three-way handshaking to establish the connection

Three-way handshaking is done by sending a connection request to the destination. At the destination side, if a connection request can be accepted, the connection is granted, and also a connection request is transmitted from destination to source. The source finally acknowledges the connection request from the destination and indicates that the connection is ready for data transfer. The connection termination also similarly ends with a three-way handshake. Due to this connection setup mechanism, TCP is called a connection oriented internet protocol.

TCP is also considered a reliable protocol for data transfer, contrary to UDP. This reliability is achieved by ensuring in-order transfer of data, acknowledgments, flow control, and congestion control mechanisms.

TCP is a very vast topic, but we'll not cover everything in this project. In the next chapter, we'll go through the structure of the TCP datagram and its flags, which is the main topic of our project.

Chapter 2: TCP Packets

2.1 Introduction

In the previous chapter, we briefly introduced computer networks and TCP. This chapter will delve into more specific parts of our project, i.e., TCP packets and Flags. We'll go through various parts of the TCP packet and discuss what each flag is used for. From the next chapter onwards, we'll discuss how our project analyses TCP packets and their flags.

2.2 TCP Segment/Packets

The TCP segment consists of header fields and a data field. The data field contains a chunk of application data. The MSS limits the maximum size of a segment's data field. When TCP sends a large file, such as an image, as part of a Web page, it typically breaks the file into chunks of size MSS (except for the last chunk, which will often be less than the MSS). The MSS is typically set by first determining the length of the largest linklayer frame that the local sending host can send (the so-called maximum transmission unit, MTU), and then setting the MSS to ensure that a TCP segment (when encapsulated in an IP datagram) plus the TCP/IP header length (typically 40 bytes) will fit into a single link-layer frame.

TCP header size is typically 20 bytes and includes the following fields:

• Source and Destination Port Numbers: Used for multiplexing/ demultiplexing data from/ to upper-layer applications.

• 32-bit Sequence Number and 32-bit Acknowledgement number: Used for in-order and reliable data transfer between source and destination.

• 16-bit Receive window: It is mainly used for the flow control mechanism. It is used to indicate the number of bytes the receiver is willing to accept.

• 4-bit Header Length: It is used to specify the length of the header in 32-bit words. TCP header can be of variable length due to the TCP options field.

(Typically, the Options field is empty, so that the length of the typical TCP header is 20)

• Variable-length Options field: The Options field is used when a sender and receiver negotiate the maximum segment size (MSS) or as a window scaling factor for use in high-speed networks. A time-stamping option is also defined.

• 6-bit Flags: (Will be discussed in next section.)

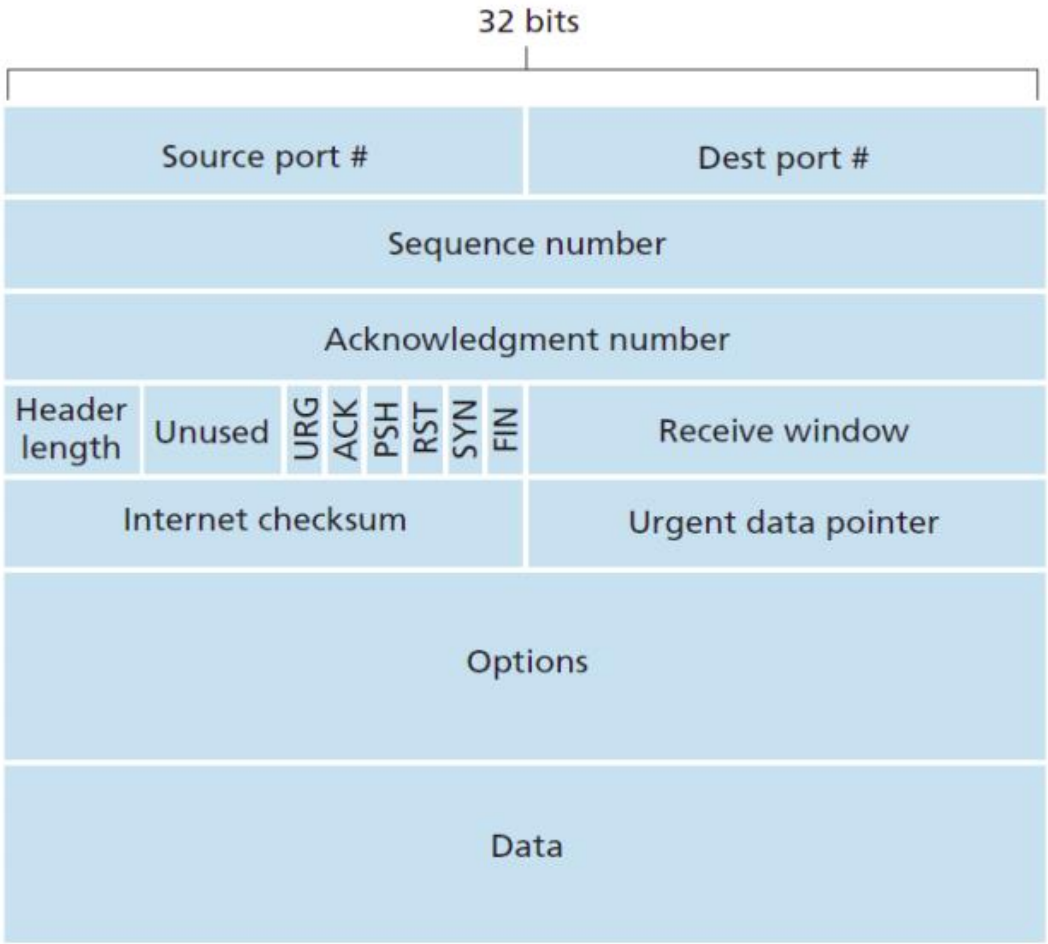


Fig 2.1: TCP Segment Structure

Chapter 3: Introduction to UDP

3.1 Connectionless Transfer (User Datagram Protocol)

UDP takes messages from the application process, attaches source and destination port number fields, adds two other small fields, and passes the resulting segment to the network layer. The network layer encapsulates the transport-layer segment into an IP datagram and then makes a best-effort attempt to deliver the segment to the receiving host. If the segment arrives at the receiving host, UDP uses the destination port number to deliver the segment's data to the correct application process.

The Application data occupies the data field of the UDP segment. The UDP header has four fields, each consisting of two bytes. The port numbers allow the destination host to pass the application data to the correct process running on the destination end system. The length field specifies the number of bytes in the UDP segment (header plus data). The checksum is used by the receiving host to check whether errors have been introduced into the segment.

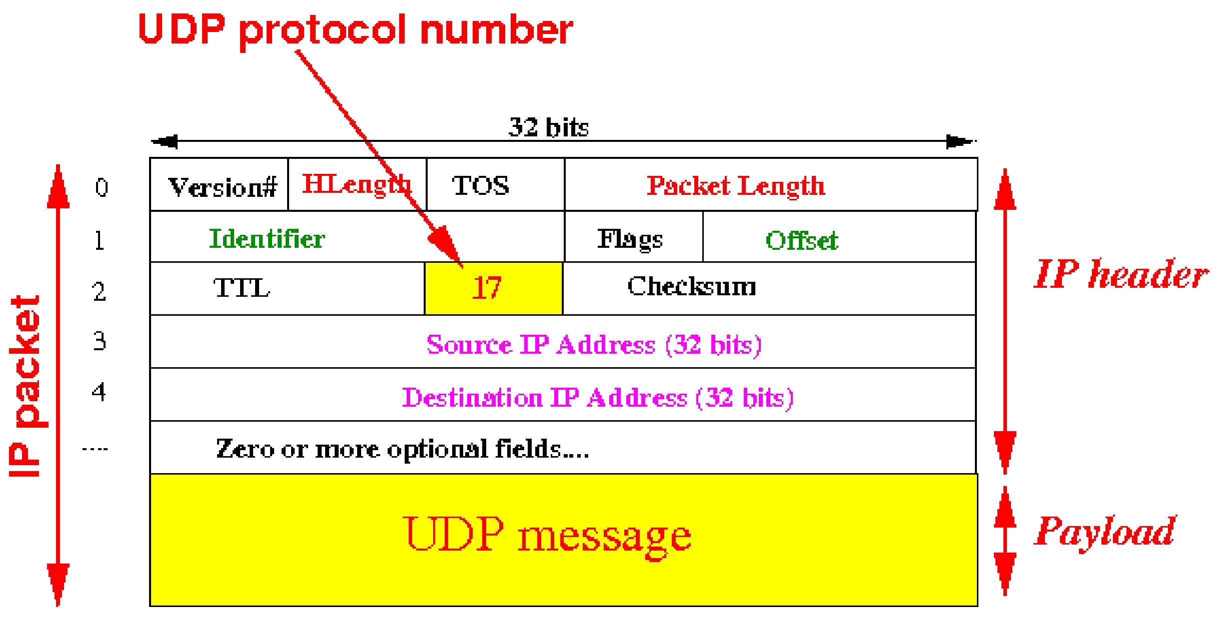


Fig 3.1: UDP Segment Structure

Chapter 4: Network Statistics

4.1 Round Trip Time (RTT)

The round-trip time for a segment is the amount of time between when the segment is sent and when an acknowledgement for the segment is received.

4.2 Throughput

The throughput is the amount of data per second that can be transferred between end systems.

4.3 Processing Delay

The processing delay is the time required to examine the packet's header and determine where to direct the packet.

4.4 Queueing Delay

The queueing delay is the amount of time a packet has to wait before being transmitted onto the link.

4.5 Transmission Delay

The transmission delay is the amount of time required to transmit all of the packet's bits into the link.

4.6 Propagation Delay

The propagation delay is the time required to propagate from the beginning of the link to the end router.

Chapter 5: Implementation in Code

5.1 Introduction

We have developed a command line interface which analyses packets present in a pcap file and present the network statistics such as throughput, average round trip time and transmission speed, for those packets.

This pcap file can downloaded or generated live via any packet sniffing tool such as Wireshark and then be fed as input to our interface.

We have also added a feature giving the user the liberty to decide how many packets the user wants to sniff from the file.

After displaying all packets, its protocols, payload, and respective details, we display the network statistics in the output.

5.2 GitHub Link for the Code

The code has been completed and tested with .pcap and .pcapng files the samples of which along with the source code file has been uploaded in a GitHub repository we have created, the link to which is:

https://github.com/aiqqia/Network-Statistics-Computer-Networks-MiniProject

5.3 Walking through the code

5.3.1 Header Files and Libraries used:

The interface was made in the C language. The following header files were used for the implementation:

- stdio.h
- pcap.h

- stdlib.h
- netinet/in.h
- netinet/tcp.h
- netinet/udp.h
- netinet/ip.h
- unistd.h
- net/ethernet.h
- string.h

Using libpcap allows us to capture or send packets from a live network device or a file. These code examples will walk us through using libpcap to find network devices, get information about devices, process packets in real time or offline, send packets, and even listen to wireless traffic. This is aimed at Debian based Linux distributions but may also work on Mac OSX. Not intended for Windows, but WinPcap is a port that is available. Compiling a pcap program requires linking with the pcap lib. We can install it in Debian based distributions with:

```
sudo apt-get install libpcap-dev
```

Once the libpcap dependency is installed, we can compile pcap programs with the following command. We will need to run the program as root or with sudo to have permission to access the network card:

```
gcc <filename> -lpcap
```

The code we have written to include the respective header files and libraries used is shown in the screenshot of the snippet of the code below:

```c
#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>
#include <assert.h>

#define APP_NAME "sniffex"
#define APP_DESC "Sniffer example using libpcap"

/* 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
```

We are including all header files respective to the functions and header structures we have used in our program.

5.3.2 Structures defined for sniffing:

```c
/* Ethernet header */
You, 8 hours ago | 1 author (You)
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 */
You, 8 hours ago | 1 author (You)
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;

You, 8 hours ago | 1 author (You)
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 */
};

You, 8 hours ago | 1 author (You)
struct sniff_udp {
  u_short sport; //source port
  u_short dport; //destination port
  u_short len; //datagram length
  u_short crc; //checksum
};
```

The defined structures and the purpose they serve are mentioned in the comments of the code as shown in the code screenshot above.

The data in structures are relevant to what we read for TCP and UDP headers and packets. Thus, the structures are well defined and explained through the variables used and the comments given int the code.

5.3.3 Prototyping functions required for the code:

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

void print_payload(const u_char * payload, int len);

void print_hex_ascii_line(const u_char * payload, int len, int offset);

void print_app_usage(void);
```

5.3.4 Functions for printing payload of the packets sniffed:

```c
/*
 * print help text
 */
void print_app_usage(void) {

  printf("Usage: %s [interface]\n", APP_NAME);
  printf("\n");
  printf("Options:\n");
  printf("    interface    Listen on <interface> for packets.\n");
  printf("\n");
  return;
}

/*
 * print data in rows of 16 bytes: offset   hex   ascii
 *
 * 00000   47 45 54 20 2f 20 48 54  54 50 2f 31 2e 31 0d 0a   GET / HTTP/1.1..
 */
void print_hex_ascii_line(const u_char * payload, int len, int offset) {

  int i;
  int gap;
  const u_char * ch;

  /* offset */
  printf("%05d   ", offset);

  /* hex */
  ch = payload;
  for (i = 0; i < len; i++) {
    printf("%02x ", * ch);
    ch++;
    /* print extra space after 8th byte for visual aid */
    if (i == 7)
      printf(" ");
  }
```

```c
  /* print space to handle line less than 8 bytes */
  if (len < 8)
    printf(" ");

  /* fill hex gap with spaces if not full line */
  if (len < 16) {
    gap = 16 - len;
    for (i = 0; i < gap; i++) {
      printf("   ");
    }
  }
  printf("   ");

  /* ascii (if printable) */
  ch = payload;
  for (i = 0; i < len; i++) {
    if (isprint( * ch))
      printf("%c", * ch);
    else
      printf(".");
    ch++;
  }

  printf("\n");

  return;
}

/*
 * print packet payload data (avoid printing binary data)
 */
void print_payload(const u_char * payload, int len) {

  int len_rem = len;
  int line_width = 16; /* number of bytes per line */
  int line_len;
  int offset = 0; /* zero-based offset counter */
```

```
    const u_char * ch = payload;

    if (len <= 0)
        return;

    /* data fits on one line */
    if (len <= line_width) {
        printf("  \t\t");
        print_hex_ascii_line(ch, len, offset);
        return;
    }

    /* data spans multiple lines */
    while(1) {
        /* compute current line length */
        line_len = line_width % len_rem;
        /* print line */
        printf("  \t\t");
        print_hex_ascii_line(ch, line_len, offset);
        /* compute total remaining */
        len_rem = len_rem - line_len;
        /* shift pointer to remaining bytes to print */
        ch = ch + line_len;
        /* add offset */
        offset = offset + line_width;
        /* check if we have line width chars or less */
        if (len_rem <= line_width) {
            /* print last line and get out */
            printf("  \t\t");
            print_hex_ascii_line(ch, len_rem, offset);
            break;
        }
    }

    return;
}
```

The functions “print_payload()” and “print_hex_asci_line()” are both functions which are used to help print the payload of the packets which we are sniffing from the pcap file fed as input to the program. These files use the header information and other formatting details to produce the payload which we then print individually for each packet sniffed.

5.3.5 Function for sniffing and displaying each packet individually:

got_packet is a function used as the call-back for our pcap_loop function. It takes in three arguments, a u_char pointer which is passed in the user argument to pcap_loop(), a const struct pcap_pkthdr pointer pointing to the packet timestamp and lengths, and a const u_char pointer to the first caplen (as given in the struct pcap_pkthdr a pointer to which is passed to the callback routine) bytes of data from the packet. The struct pcap_pkthdr and the packet data are not to be freed by the callback routine, and are not guaranteed to be valid after the callback routine returns; if the code needs them to be valid after the callback, it must make a copy of them. Count is used for keeping a count of packets.

ethernet is a pointer to sniff_ethernet struct, which points to the starting of the packet. ip is a pointer to sniff_ip struct, which points to the start of the packet with an offset equal to the size of the ethernet header. tcp is a pointer to sniff_tcp which points to the start of the packet with an offset of ethernet size, ip size combined. We then get to know which type of protocol is being used with the help of the switch case. Our program will continue only if the packet follows TCP or UDP protocol. For UDP, we print the source port number, destination port number and datagram length. For TCP, we print the source port number, destination port number, packet capture length, packet total length, sequence number, acknowledgement number and payload, if any.

```
/*
 * dissect/print 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 */
  const char * payload; /* Packet payload */

  int size_ip;
  int size_tcp;
  int size_payload;

  printf("\t\t-------- PACKET [Number : %d] --------\n\n", 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("**\t\tInvalid IP header length: %u bytes\n", size_ip);
    return;
  }

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

```c

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

    /* define/compute tcp header offset */
    if (ip -> ip_p == IPPROTO_UDP) {
      struct sniff_udp * udp;
      udp = (struct sniff_udp * )(packet + SIZE_ETHERNET + size_ip);
      printf("**\t\tSource Port : %u\n**\t\tDestination Port : %u\n", udp -> sport, udp -> dport);
      printf("**\t\tUDP Datagram Length : %u\n", udp -> len / 256);
    } else {

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

```c

      printf("**\t\tSource port : %d\n", ntohs(tcp -> th_sport));
      printf("**\t\tDestination port : %d\n", ntohs(tcp -> th_dport));

      printf("**\t\tPacket capture length : %d\n", header -> caplen);
      printf("**\t\tPacket total length : %d\n", header -> len);

      printf("**\t\tSequence number : %u\n", tcp -> th_seq);
      printf("**\t\tAcknowledgement number(Ack) : %u\n", tcp -> th_ack);

      /* define/compute tcp payload (segment) offset */
      payload = (u_char * )(packet + SIZE_ETHERNET + size_ip + size_tcp);

      /* compute tcp payload (segment) size */
      size_payload = ntohs(ip -> ip_len) - (size_ip + size_tcp);

      /*
       * Print payload data; it might be binary, so don't just
       * treat it as a string.
       */
      if (size_payload > 0) {
        printf("**\t\tPayload (%d bytes) : \n\n", size_payload);
        print_payload(payload, size_payload);
      }
    }
    printf("\n\t\t----------------------------------------\n\n");
    return;
  }

```

5.3.6 Function for parsing the file to give final network statistics output:

We use this function to Parse our data.txt file and get all the valuable information from it. FILE* f points to the file which we want to open (data.txt). buf array is used to store the whole line of our input, whereas temp array is used to store the value of our data, such as average speed, average packet size, average packet rate, etc. If we open the data.txt file, we will soon come to know that the eleventh and twelfth lines of the file hold the average speed of our network, the thirteenth line holds the average packet size, and the fourteenth line holds the average packet rate. The count variable is used to calculate the line number we have parsed so far. Unit conversions are done accordingly to have a standardized way of presenting our data. In the end, we calculate the average RTT of our network and output it accordingly.

Function: parse()

```c

void parse() {
    FILE * f;
    f = fopen("data.txt", "r");
    char buf[256];
    char temp[256];
    int count = 0;
    int i = 0, j = 0;
    float size, speed, Mbps, prate;
    for (int count = 0; fgets(buf, sizeof(buf), f) != NULL && count < 15; count++) {
        if (count < 11) {
            continue;
        }
        i = 0, j = 0;
        while (!isdigit(buf[i])) i++;
        while (buf[i] != ' ') {
            if (buf[i] != ',') {
                temp[j++] = buf[i];
            }
            i++;
        }
        temp[j++] = 0;

        if (count == 11) {
            speed = atof(temp);

            while(buf[i++] != ' ');
            char type[10];
            int k = 0;
            while(isalpha(buf[i]) || buf[i] == '/'){
                type[k++] = buf[i++];
            }
            type[k++] = 0;
            if(strcmp(type, "kBps") == 0) speed /= 1024.0f;
```

```c
            else if(strcmp(type, "bytes/s") == 0) speed /= 1024.0f * 1024.0f;
            else if(strcmp(type, "MBps") != 0) puts(type), assert(0);

            printf("**\t\tAVERAGE SPEED(MBps)    : %4.2f MBps\n", speed);
        } else if (count == 12) {
            Mbps = atof(temp);

            while(buf[i++] != ' ');
            char type[10];
            int k = 0;
            while(isalpha(buf[i])){
                type[k++] = buf[i++];
            }
            type[k++] = 0;
            if(strcmp(type, "kbps") == 0) Mbps /= 1024.0f;
            else if(strcmp(type, "Mbps") != 0) assert(0);

            printf("**\t\tAVERAGE SPEED(Mbps)    : %4.2f Mbps\n", Mbps);
        } else if (count == 13) {
            size = atof(temp);
            printf("**\t\tAVERAGE PACKET SIZE    : %4.2f bytes\n", size);
        } else if (count == 14) {
            prate = atof(temp);
            printf("**\t\tAVERAGE PACKET RATE/s : %4.2f kpackets/s\n", prate);
        }
    }
    printf("**\t\tAVERAGE RTT            : %f seconds\n", size * 2 / (speed * (1 << 20)));
}

```

5.3.7 Function for displaying simple terminal-based GUI:

```c

void menu() {
  system("clear");
  printf("\n********************************* MENU ***********************************");
  printf("\n\n**     Enter the file name with the .pcap extension for analysis        **");
  printf("\n\n**       ==>   ");
}

```

We have made a small terminal-based interface and to beautify, we have used this function and given proper spacing.

5.3.8 The main function that ties all the program together:

To obtain the bare-bones statistics of the network, we used the capinfos software, and stored the fetched data into a text file called “data.txt”. We have then sent this file to be parsed and statistics to be retrieved through the parse() function call mentioned in the code.

We also obtained the IP header information and the TCP/UDP header information.

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

  char filter_exp[] = "ip"; /* filter expression [3] */
  struct bpf_program fp; /* compiled filter program (expression) */
  bpf_u_int32 mask = 0; /* subnet mask */
  bpf_u_int32 net = 0; /* ip */
  int num_packets = 0; /* number of packets to capture */

  char fileName[100];
  menu();
  scanf("%s", fileName);
  handle = pcap_open_offline(fileName, errbuf);
  if (handle == NULL) {
    fprintf(stderr, "Couldn't open device\n");
    exit(EXIT_FAILURE);
  }
  printf("\n**************************************************************************");
  printf("\n\n**     Enter number of packets to be sniffed (Enter 0 for all):         **");
  printf("\n\n**     ==>   ");
  scanf("%d", & num_packets);
  printf("\n**************************************************************************");
  if (num_packets == 0)
    printf("\n\n**     Number of packets : All\n");
  else
    printf("\n\n**     Number of packets : %d\n", num_packets);

  /* make sure we're capturing on an Ethernet device [2] */
  if (pcap_datalink(handle) != DLT_EN10MB) {
    fprintf(stderr, "Not an Ethernet\n");
    exit(EXIT_FAILURE);
  }
  printf("\n********************************************************************\n\n");

  /* 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);

  char command[200];
  sprintf(command, "capinfos %s > ./data.txt", fileName);
  system(command);
  printf("\n\n******************** FINAL NETWORK STATISTICS ********************\n\n");
  parse();
  printf("\n\n****************************************************************\n\n");

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

  return 0;
}
```

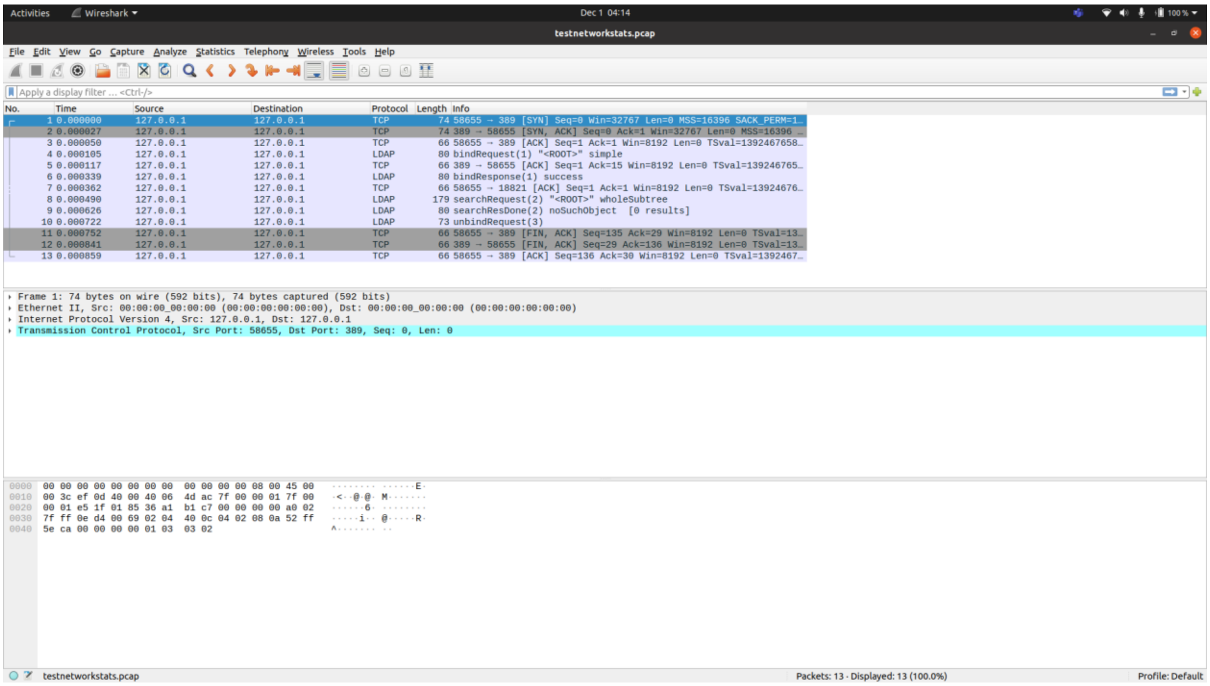
We have used some print statements to beautify the code in the end to compensate for a functioning frontend.

The overall code works and has been tested with multiple pcap and pcapng files and the order of working is that initially on running the code, we are asked to input the name of the pcap file to be fed as input. On inputting a valid file name, we can enter how many packets we want the program to sniff, and we get the appropriate display as chosen. In the end, after all the individual packets have been displayed, we can see that the network statistics are displayed.

We have verified the correctness of the statistics by checking network statistics in the Wireshark tool as well.

5.3.9 Outputs of the Program:

The first sample .pcap file is called “testnetworkstats.pcap” and the Wireshark screenshot is attached below:



We can see the packets available to be sniffed in the screenshot above and now we will run the code in the gcc compiler.

The menu driven GUI looks as follows:

We are required to input the Filename and the number of packets we want to be sniffed.

```
******************************** MENU **********************************

**     Enter the file name with the .pcap extension for analysis       **

**       ==>   testnetworkstats.pcap

*************************************************************************

**     Enter number of packets to be sniffed (Enter 0 for all):        **

**     ==>   0

*************************************************************************

**     Number of packets : All

*************************************************************************

               -------- PACKET [Number : 1] --------

**             From IP : 127.0.0.1
**             To IP : 127.0.0.1
**             Protocol : TCP
**             Source port : 58655
**             Destination port : 389
**             Packet capture length : 74
**             Packet total length : 74
**             Sequence number : 3350307126
**             Acknowledgement number(Ack) : 0

               ----------------------------------------

               -------- PACKET [Number : 2] --------

**             From IP : 127.0.0.1
**             To IP : 127.0.0.1
**             Protocol : TCP
**             Source port : 389
**             Destination port : 58655
**             Packet capture length : 74
**             Packet total length : 74
**             Sequence number : 750250550
**             Acknowledgement number(Ack) : 3367084342
```

```
                ----------------------------------------

                -------- PACKET [Number : 3] --------

**              From IP : 127.0.0.1
**              To IP : 127.0.0.1
**              Protocol : TCP
**              Source port : 58655
**              Destination port : 389
**              Packet capture length : 66
**              Packet total length : 66
**              Sequence number : 3367084342
**              Acknowledgement number(Ack) : 767027766

                ----------------------------------------

                -------- PACKET [Number : 4] --------

**              From IP : 127.0.0.1
**              To IP : 127.0.0.1
**              Protocol : TCP
**              Source port : 58655
**              Destination port : 389
**              Packet capture length : 80
**              Packet total length : 80
**              Sequence number : 3367084342
**              Acknowledgement number(Ack) : 767027766
**              Payload (14 bytes) :

                00000   30 0c 02 01 01 60 07 02  01 03 04 00 80 00          0....`........

                ----------------------------------------

                -------- PACKET [Number : 5] --------

**              From IP : 127.0.0.1
**              To IP : 127.0.0.1
**              Protocol : TCP
**              Source port : 389
**              Destination port : 58655
**              Packet capture length : 66
**              Packet total length : 66
**              Sequence number : 767027766
**              Acknowledgement number(Ack) : 3601965366
```

```
                ----------------------------------------

                -------- PACKET [Number : 6] --------

**              From IP : 127.0.0.1
**              To IP : 127.0.0.1
**              Protocol : TCP
**              Source port : 389
**              Destination port : 58655
**              Packet capture length : 80
**              Packet total length : 80
**              Sequence number : 767027766
**              Acknowledgement number(Ack) : 3601965366
**              Payload (14 bytes) :

                00000   30 0c 02 01 01 61 07 0a  01 00 04 00 04 00          0....a........

                ----------------------------------------

                -------- PACKET [Number : 7] --------

**              From IP : 127.0.0.1
**              To IP : 127.0.0.1
**              Protocol : TCP
**              Source port : 58655
**              Destination port : 18821
**              Packet capture length : 66
**              Packet total length : 66
**              Sequence number : 3601965366
**              Acknowledgement number(Ack) : 1001908790

                ----------------------------------------

                -------- PACKET [Number : 8] --------

**              From IP : 127.0.0.1
**              To IP : 127.0.0.1
**              Protocol : TCP
**              Source port : 58655
**              Destination port : 389
**              Packet capture length : 179
**              Packet total length : 179
**              Sequence number : 3601965366
**              Acknowledgement number(Ack) : 1001908790
**              Payload (113 bytes) :
```

```
                ---------------------------------------

                -------- PACKET [Number : 8] --------

**              From IP : 127.0.0.1
**              To IP : 127.0.0.1
**              Protocol : TCP
**              Source port : 58655
**              Destination port : 389
**              Packet capture length : 179
**              Packet total length : 179
**              Sequence number : 3601965366
**              Acknowledgement number(Ack) : 1001908790
**              Payload (113 bytes) :

                00000   30 6f 02 01 02 63 6a 04  00 0a 01 02 0a 01 00 02    0o...cj.........
                00016   01 00 02 01 00 01 01 00  a9 3c 81 1c 32 2e 31 36    .........<..2.16
                00032   2e 38 34 30 2e 07 2e 31  31 33 37 33 30 2e 33 2e    .840...113730.3.
                00048   33 2e 32 2e 34 36 2e 31  82 10 64 65 70 61 72 74    3.2.46.1..depart
                00064   6d 65 6e 74 4e 75 6d 62  65 72 83 07 3e 3d 4e 34    mentNumber..>=N4
                00080   37 30 39 84 01 ff 30 19  04 02 63 6e 04 02 73 6e    709...0...cn..sn
                00096   04 0f 74 65 6c 65 70 68  6f 6e 65 4e 75 6d 62 65    ..telephoneNumbe
                00112   72                                                  r

                ---------------------------------------

                -------- PACKET [Number : 9] --------

**              From IP : 127.0.0.1
**              To IP : 127.0.0.1
**              Protocol : TCP
**              Source port : 389
**              Destination port : 58655
**              Packet capture length : 80
**              Packet total length : 80
**              Sequence number : 1001908790
**              Acknowledgement number(Ack) : 1202889014
**              Payload (14 bytes) :

                00000   30 0c 02 01 02 65 07 0a  01 20 04 00 04 00          0....e... ....

                ---------------------------------------
```

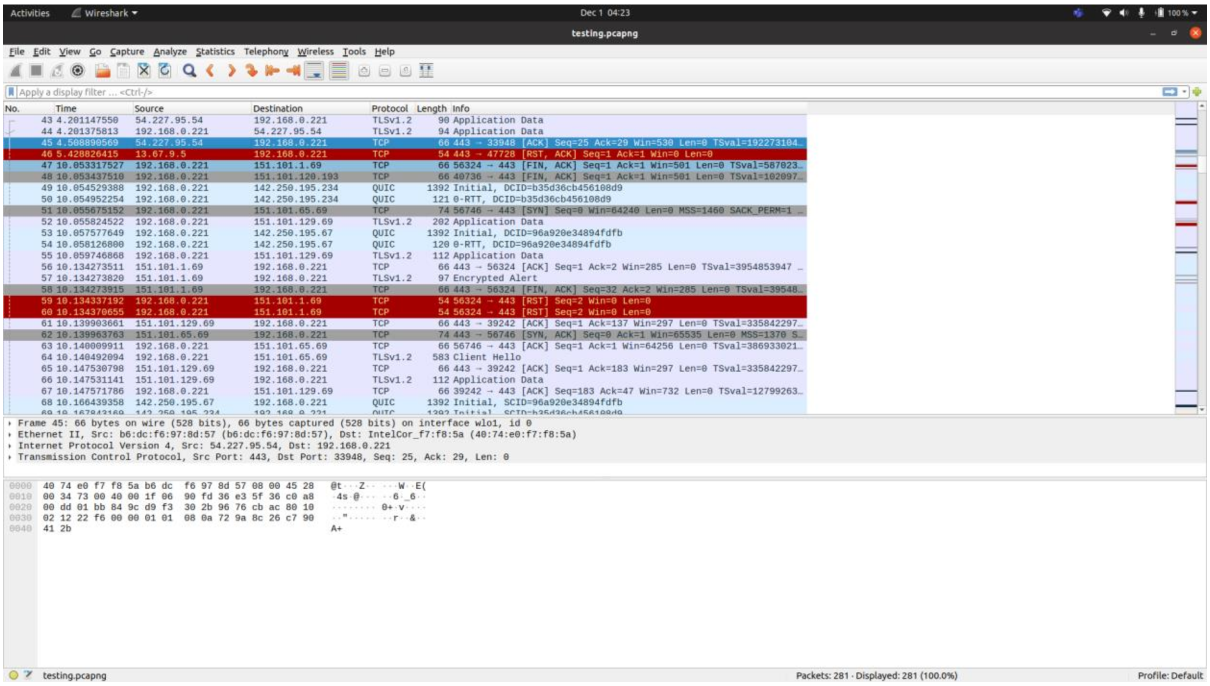
```
                -------- PACKET [Number : 10] --------

**              From IP : 127.0.0.1
**              To IP : 127.0.0.1
**              Protocol : TCP
**              Source port : 58655
**              Destination port : 389
**              Packet capture length : 73
**              Packet total length : 73
**              Sequence number : 1202889014
**              Acknowledgement number(Ack) : 1236789814
**              Payload (7 bytes) :

                00000   30 05 02 01 03 42 00                                0....B.

                ---------------------------------------

                -------- PACKET [Number : 11] --------

**              From IP : 127.0.0.1
**              To IP : 127.0.0.1
**              Protocol : TCP
**              Source port : 58655
**              Destination port : 389
**              Packet capture length : 66
**              Packet total length : 66
**              Sequence number : 1320329526
**              Acknowledgement number(Ack) : 1236789814

                ---------------------------------------

                -------- PACKET [Number : 12] --------

**              From IP : 127.0.0.1
**              To IP : 127.0.0.1
**              Protocol : TCP
**              Source port : 389
**              Destination port : 58655
**              Packet capture length : 66
**              Packet total length : 66
**              Sequence number : 1236789814
**              Acknowledgement number(Ack) : 1337106742
```

```
                ---------------------------------------

                -------- PACKET [Number : 13] --------

**              From IP : 127.0.0.1
**              To IP : 127.0.0.1
**              Protocol : TCP
**              Source port : 58655
**              Destination port : 389
**              Packet capture length : 66
**              Packet total length : 66
**              Sequence number : 1337106742
**              Acknowledgement number(Ack) : 1253567030

                ---------------------------------------



********************** FINAL NETWORK STATISTICS **********************

**              AVERAGE SPEED(MBps)    : 1.18 MBps
**              AVERAGE SPEED(Mbps)    : 9.42 Mbps
**              AVERAGE PACKET SIZE    : 79.69 bytes
**              AVERAGE PACKET RATE/s : 15.00 kpackets/s
**              AVERAGE RTT            : 0.000129 seconds


***********************************************************************


Capture complete.

manan@manan-Inspiron-7591:~/Documents/C_Programs$
```

In this final screenshot, we can notice that after all thirteen packets are displayed, we are able to see the network statistics displayed which is the outcome required for our project.

To test our final project, we have used another sample input file called “testing.pcapng” which is very huge in comparison, and we can see the packets and network statistics for that too as shown below:

Initially, we show the Wireshark screenshot for the given pcap file:



The menu driven inputs:

```
********************************* MENU ***********************************

**     Enter the file name with the .pcap extension for analysis        **

**       ==>    testing.pcapng

**************************************************************************

**     Enter number of packets to be sniffed (Enter 0 for all):         **

**     ==>    0

**************************************************************************

**     Number of packets : All

**************************************************************************

                -------- PACKET [Number : 1] --------

**              From IP : 157.240.192.52
**              To IP : 192.168.0.221
**              Protocol : TCP
**              Source port : 443
**              Destination port : 36576
**              Packet capture length : 99
**              Packet total length : 99
**              Sequence number : 1953468210
**              Acknowledgement number(Ack) : 2926838274
**              Payload (33 bytes) :

                00000   17 03 03 00 1c 4c ae 01  dd 40 98 bd dc 32 83 d4    .....L...@...2..
                00016   42 f6 ba 43 e6 a1 16 c3  c2 08 9e 9d 01 ea 9c fa    B..C............
                00032   8b                                                  .

                ----------------------------------------
```

The last packet along with the network statistics as shown below:

```
                -------- PACKET [Number : 279] --------

**              From IP : 54.227.95.54
**              To IP : 192.168.0.221
**              Protocol : TCP
**              Source port : 443
**              Destination port : 33948
**              Packet capture length : 66
**              Packet total length : 66
**              Sequence number : 1127281625
**              Acknowledgement number(Ack) : 3368777366

                ----------------------------------------



********************** FINAL NETWORK STATISTICS **********************

**              AVERAGE SPEED(MBps)   : 0.01 MBps
**              AVERAGE SPEED(Mbps)   : 0.06 Mbps
**              AVERAGE PACKET SIZE   : 465.27 bytes
**              AVERAGE PACKET RATE/s : 17.00 kpackets/s
**              AVERAGE RTT           : 0.112262 seconds


***********************************************************************


Capture complete.
```

[END OF CHAPTER]

APPENDICES

GitHub link to our code and input test files:

https://github.com/aiqqia/Network-Statistics-Computer-Networks-MiniProject

Link for downloading sample pcap and pcapng files of all sizes and formats along with protocol filters required:

https://www.wireshark.org/download/automated/captures/

REFERENCES

1. Computer Networking: A Top-Down Approach, 6th Edition
2. https://www.opensourceforu.com/2011/02/capturing-packets-c-program-libpcap/
3. https://www.devdungeon.com/content/using-libpcap-c
4. http://yuba.stanford.edu/~casado/pcap/section4.html
5. https://www.wireshark.org/docs/man-pages/capinfos.html
6. https://danielmiessler.com/study/tcpdump/