

1.

a)

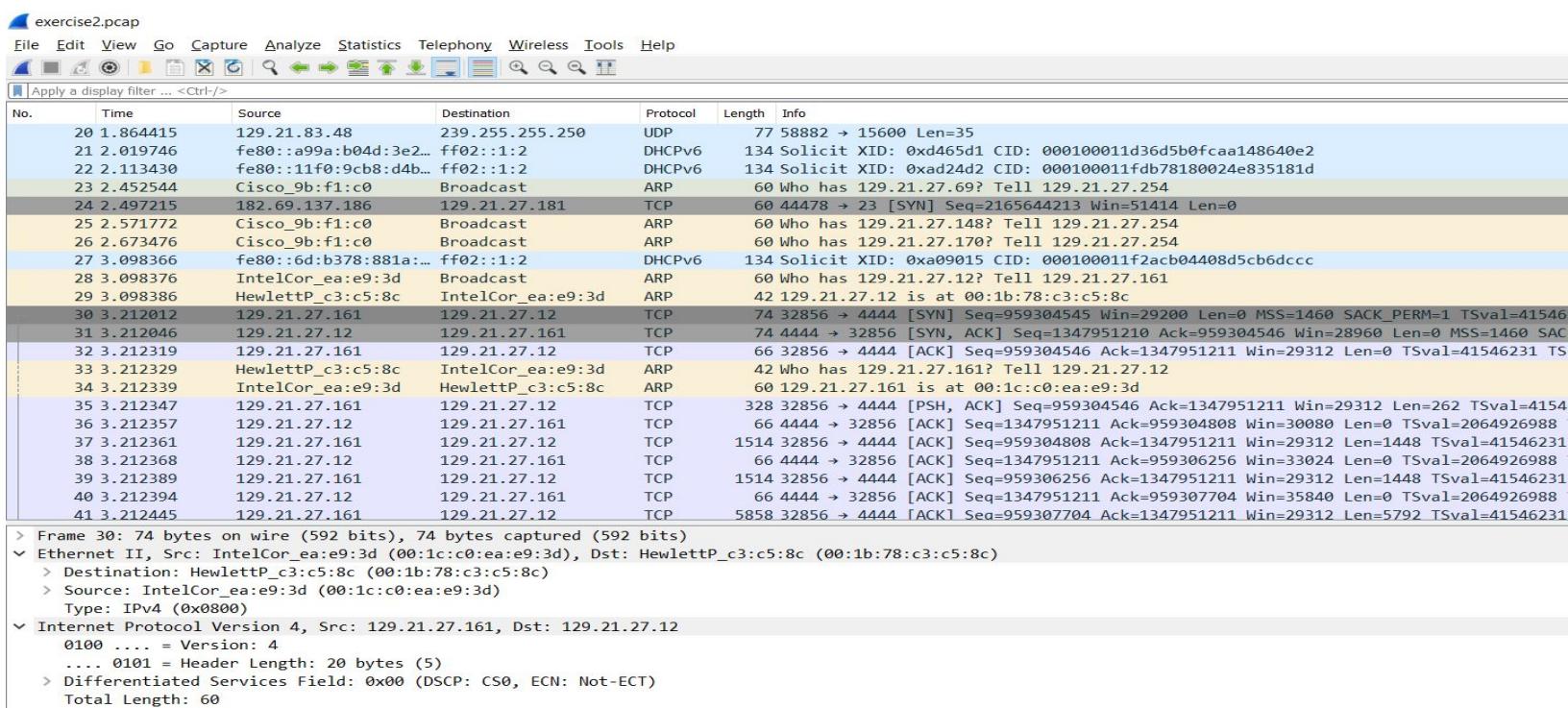


Fig 1: Ethernet II and 74 bytes on the wire.

Ethernet II mentioned in the link layer is a wired protocol used for data transfer. Thus, we can conclude that the network interface from where traffic was captured was wired (an Ethernet interface).

b)

Different packets/protocols	Layers encapsulated
DHCPv6	Physical, Data Link, Network, Transport
STP	Physical, Data Link
ARP	Physical, Data Link
UDP	Physical, Data Link, Network, Transport
TCP	Physical, Data Link, Network, Transport

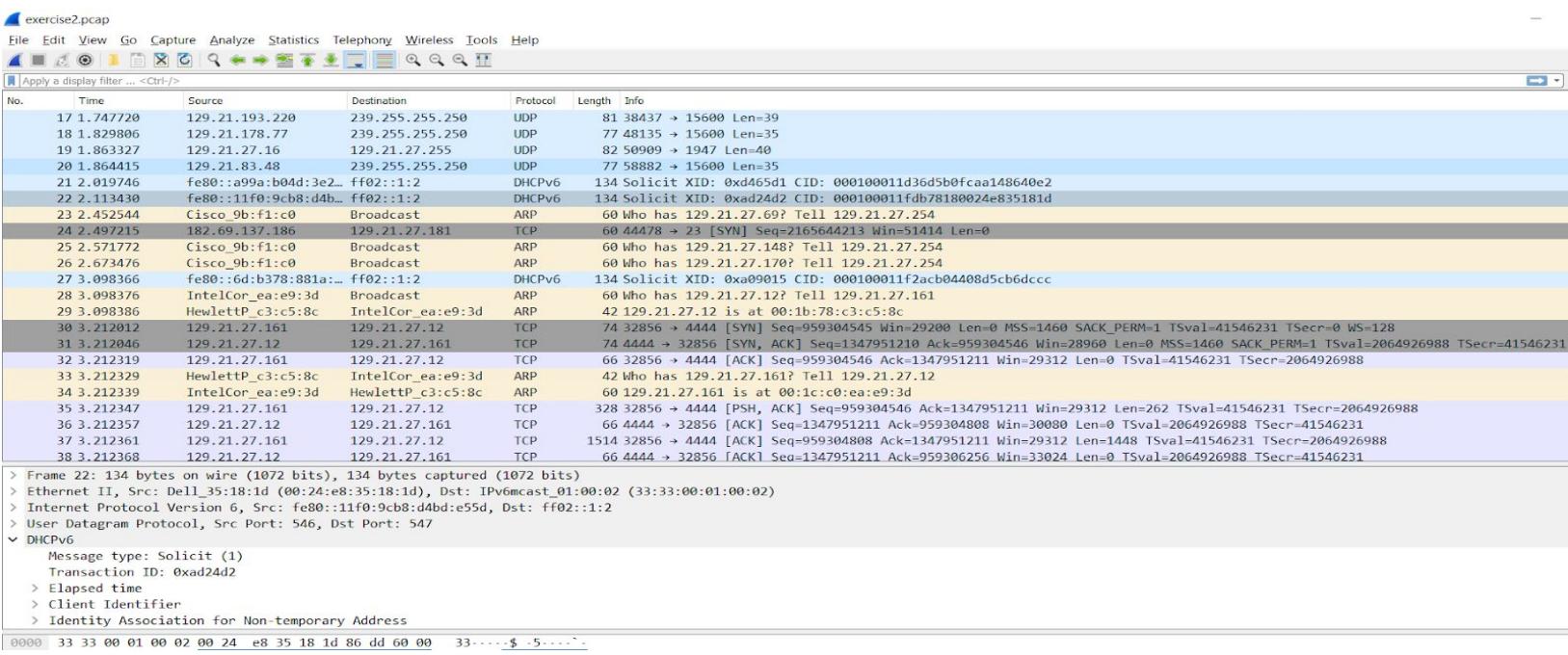


Fig 2: the layers encapsulated for packet 22, which has the DHCPv6 protocol. Check the packet detail window

From the above figure, we can interpret that the DHCPv6 protocol packet has the Physical layer, Data Link layer, Network layer, and Transport layer encapsulated.

Similarly, we can obtain this information for other packets/protocols. All the different protocols available in the used pcap file are presented in a tabular form above.

c) Packet numbers 28 and 29 follow the Address Resolution Protocol (ARP). Packet 28 is an ARP request and packet 29 is an ARP reply. Source and destination addresses are MAC addresses. In ARP exchange the sender knows the IP address of the destination but has no entry of its MAC address in its MAC table. By sending an ARP request, which is a broadcast message i.e the destination MAC address field is all '1' in the frame, IP address is the destination IP address. When the broadcast frame is received at the node whose IP address is included in the frame, it acknowledges the frame and sends an ARP reply back only to the source with its MAC address included in the frame.

In our pcap file, in packet 28, the source has MAC address 00:1c:c0:ea:e9:3d (IntelCor_ea:e9:3d) and IP address as 129.21.27.161. It sends a broadcast to all the nodes in the LAN with destination MAC address as ff:ff:ff:ff:ff:ff (Broadcast) and destination IP address as 129.21.27.12. When the node with IP address 129.21.27.12 receives the broadcast message it has acknowledged it and sends an ARP reply back to the source with its MAC address included in the frame.

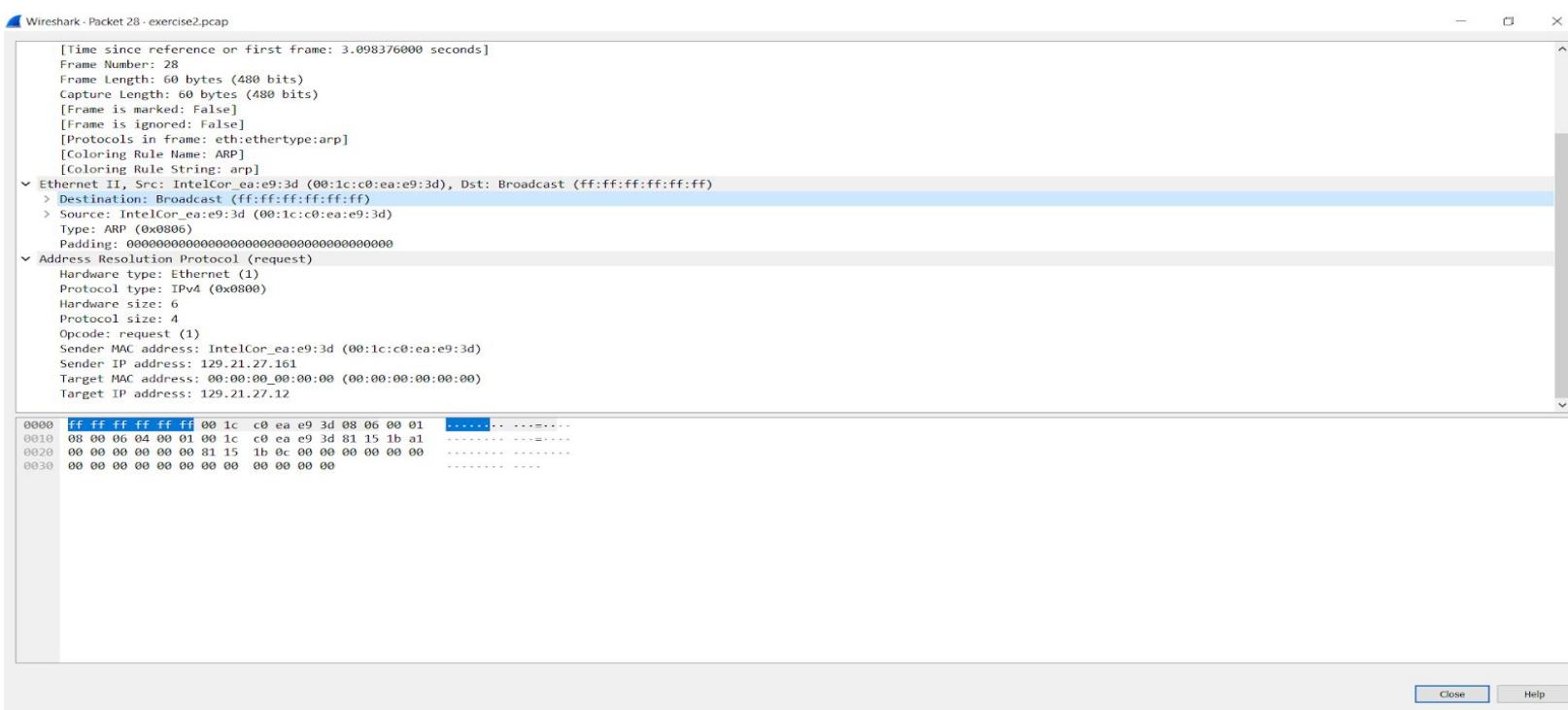


Fig 3: ARP request (Packet 28), Destination address -> MAC address ff:ff:ff:ff:ff:ff

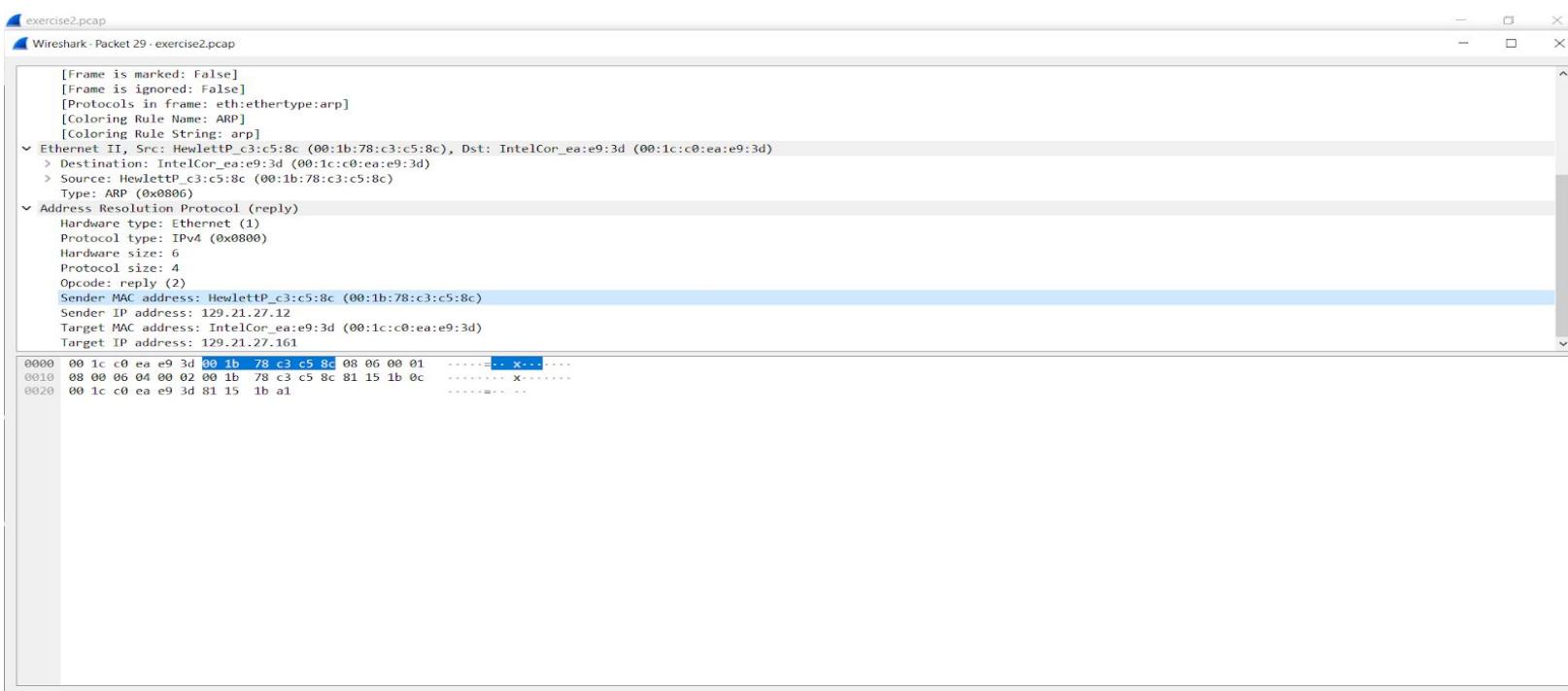


Fig 4: ARP reply (Packet 29), Destination MAC address (HewlettP) obtained.

d) The pcap file used contains a TCP connection for file transfer from source to destination. The two IP addresses involved in this exchange are 129.21.27.12 and 129.21.27.161. Packets 28, 29 and packets 33, 34 are the ARP packets used to obtain the MAC address of one another. TCP exchange starts at packet 30 where the source IP address 129.21.27.161 sends an SYN (Synchronize Sequence Number) to destination IP address 129.21.27.12 for synchronizing the exchange. The destination IP address 129.21.27.12 sends back an SYN, ACK to acknowledge this synchronization and to tell the source that it is ready for TCP exchange in the packet 31. Packet 32 denotes the ACK sent by the source to destination that confirms the reliable connection made between them and tells the destination that the transfer will start now. These three packets denote the 3-way handshake of TCP. From packets 35 to packet 82 there is data transfer from source to destination. Packet 35 includes the first payload "RFC 791". Packet 83 includes data transfer from destination to source, the data was "I got your file". From packet 84 to 87 the TCP termination takes place. In packet 84 the destination node sends a packet with FIN and ACK to the source denoting that it wants to terminate the TCP connection. Packet 85 is the ACK sent by the source node to the destination acknowledging the request. Also, in packet 86 source node sends a packet with FIN and ACK flag high to denote that graceful termination of the TCP will be done from both the end. In packet 87, the destination node sends a packet with ACK flag high and the TCP connection is terminated.

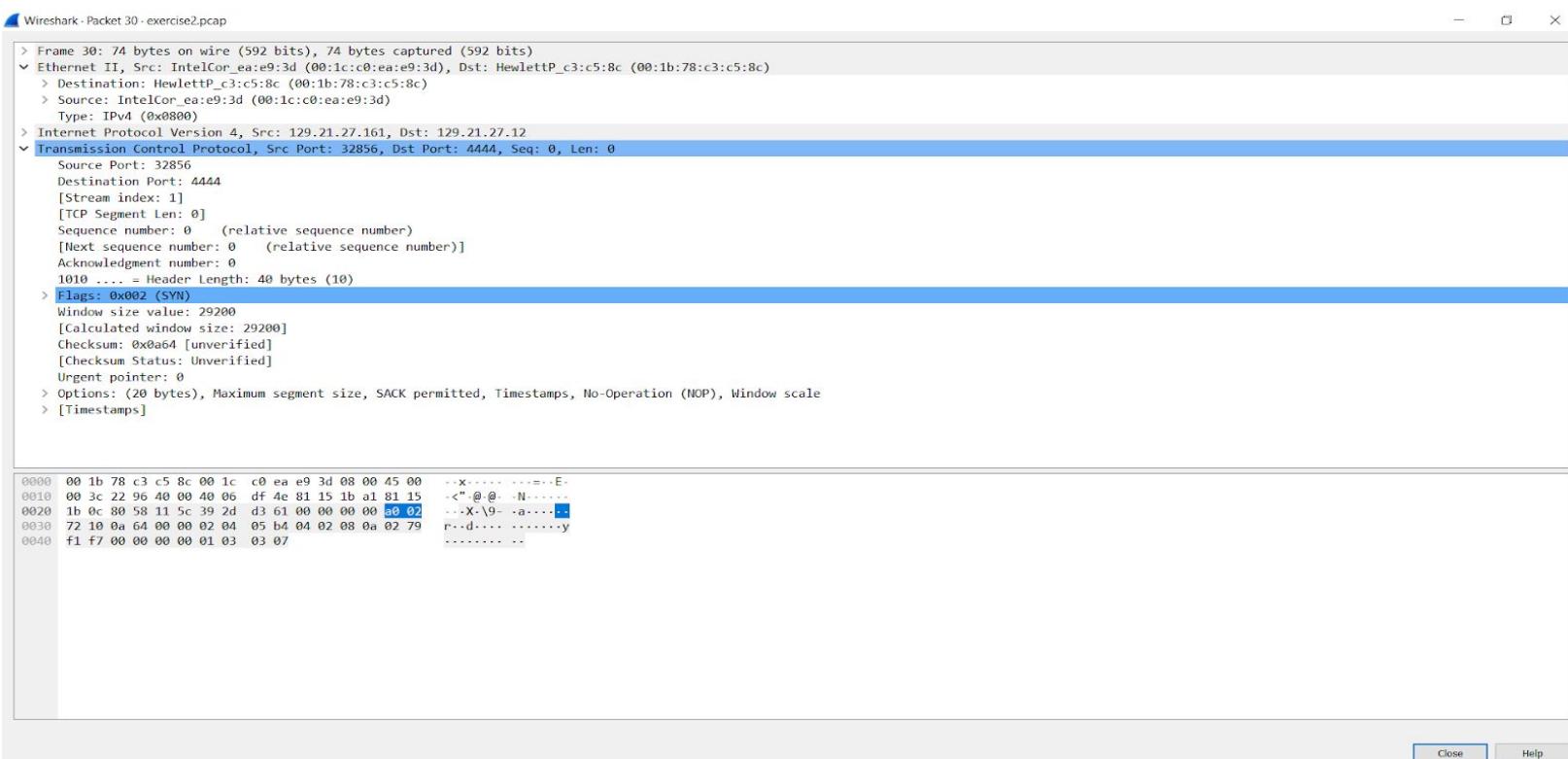


Fig 5: Packet 30, SYN flag for synchronization of sequence number between source and destination

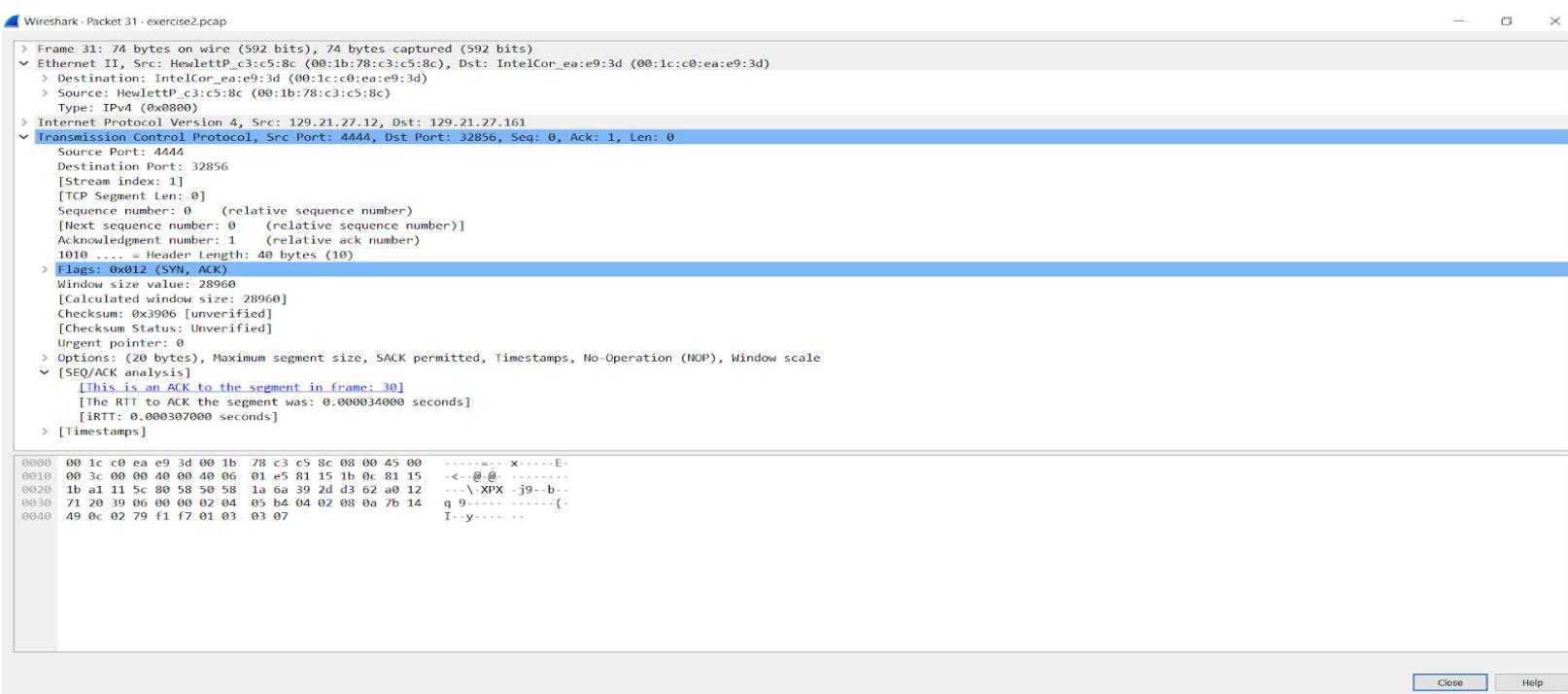


Fig 6: Packet 31, SYN+ACK received from the destination



Fig 7: Packet 32, ACK sent by the source to destination to denote the 3-way handshake complete and data transfer will begin.

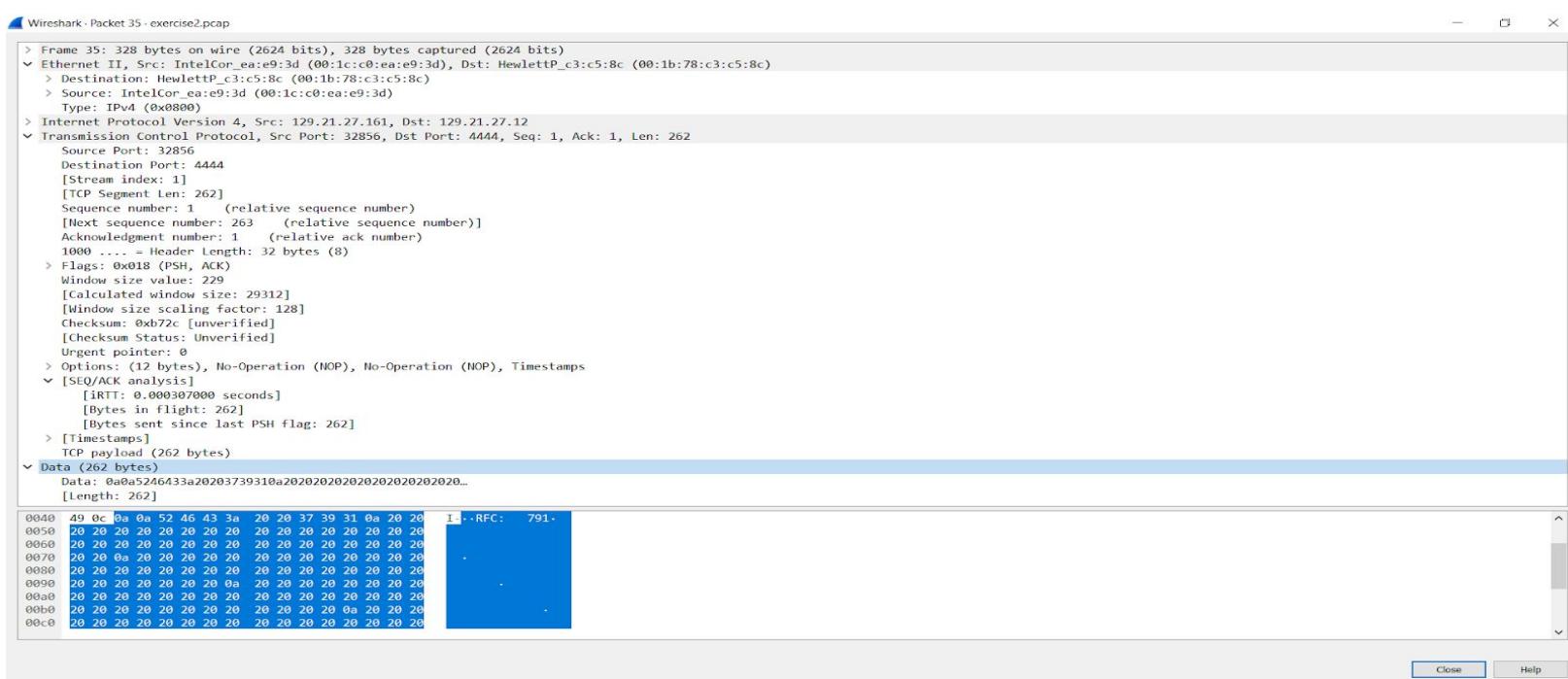


Fig 8: Packet 35, the source sends the first payload to the destination

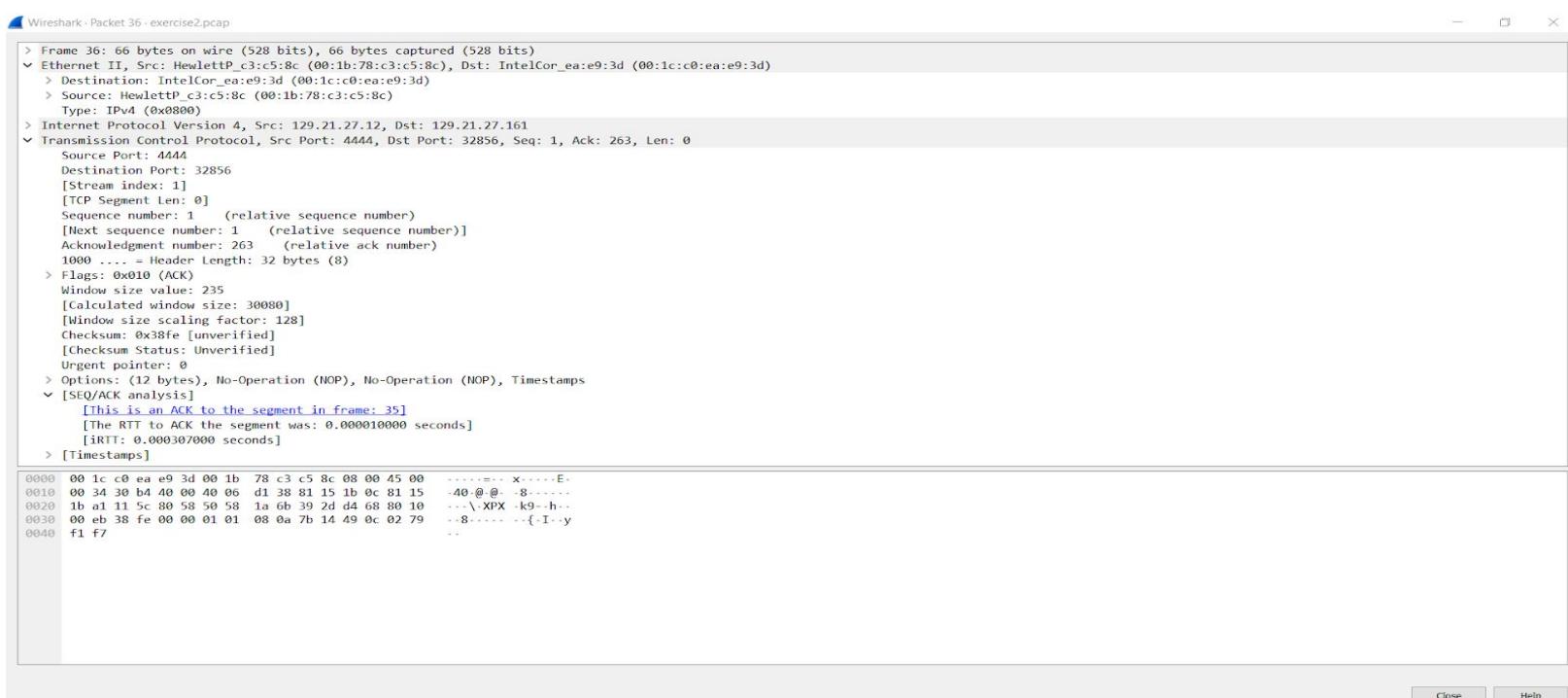


Fig 9: Packet 36. ACK sent back to the source which denotes that packet 35 data received.

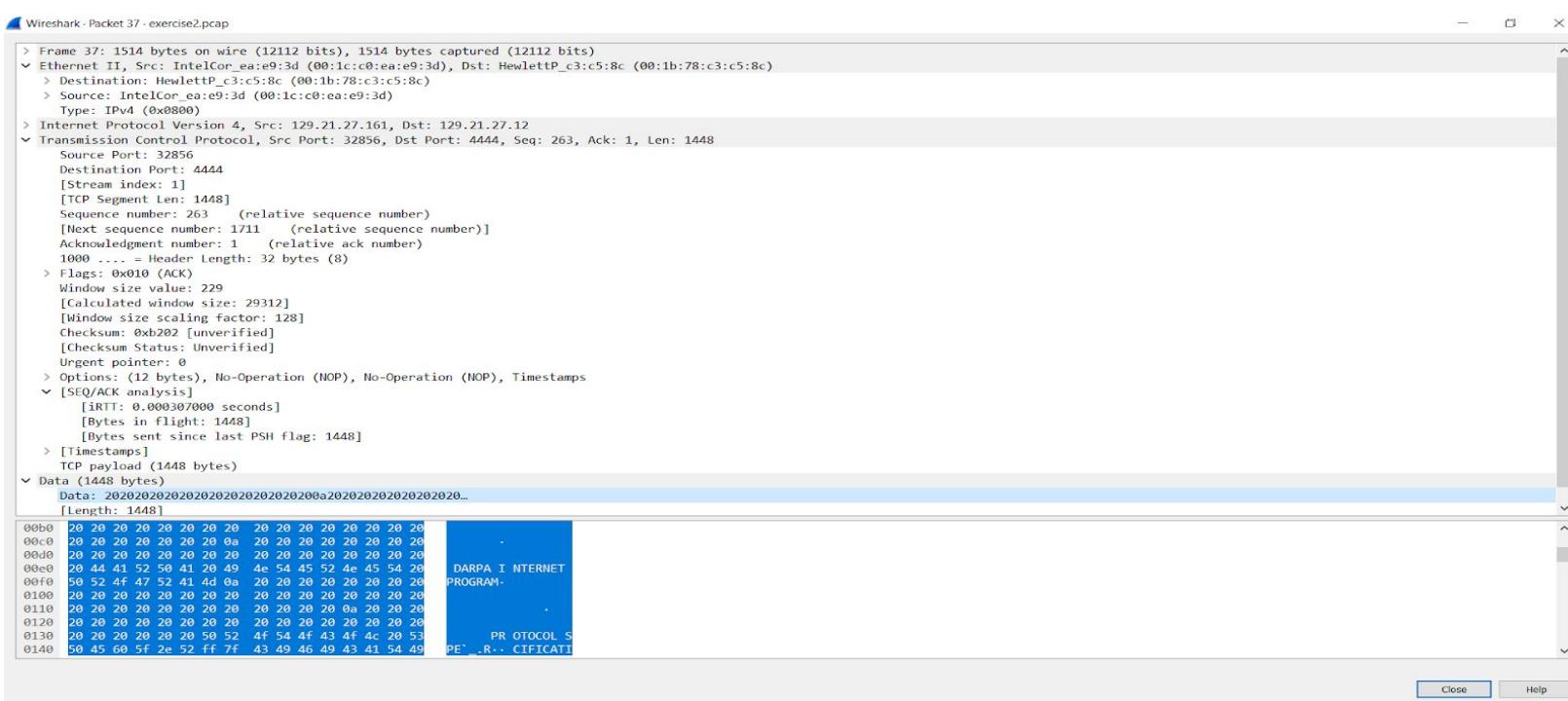


Fig 10: Packet 37, 2nd payload send to the destination

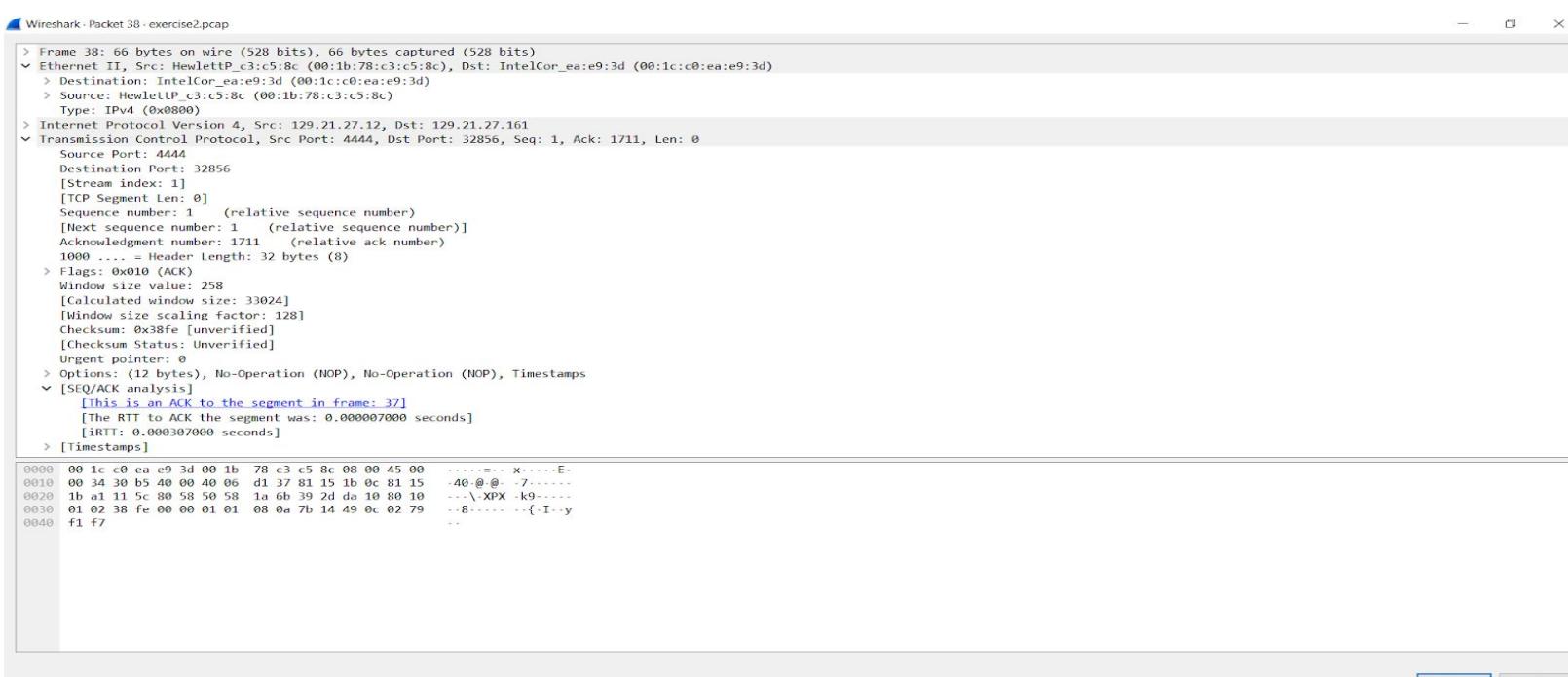


Fig 11: Packet 38, ACK for the packet 37 received correctly

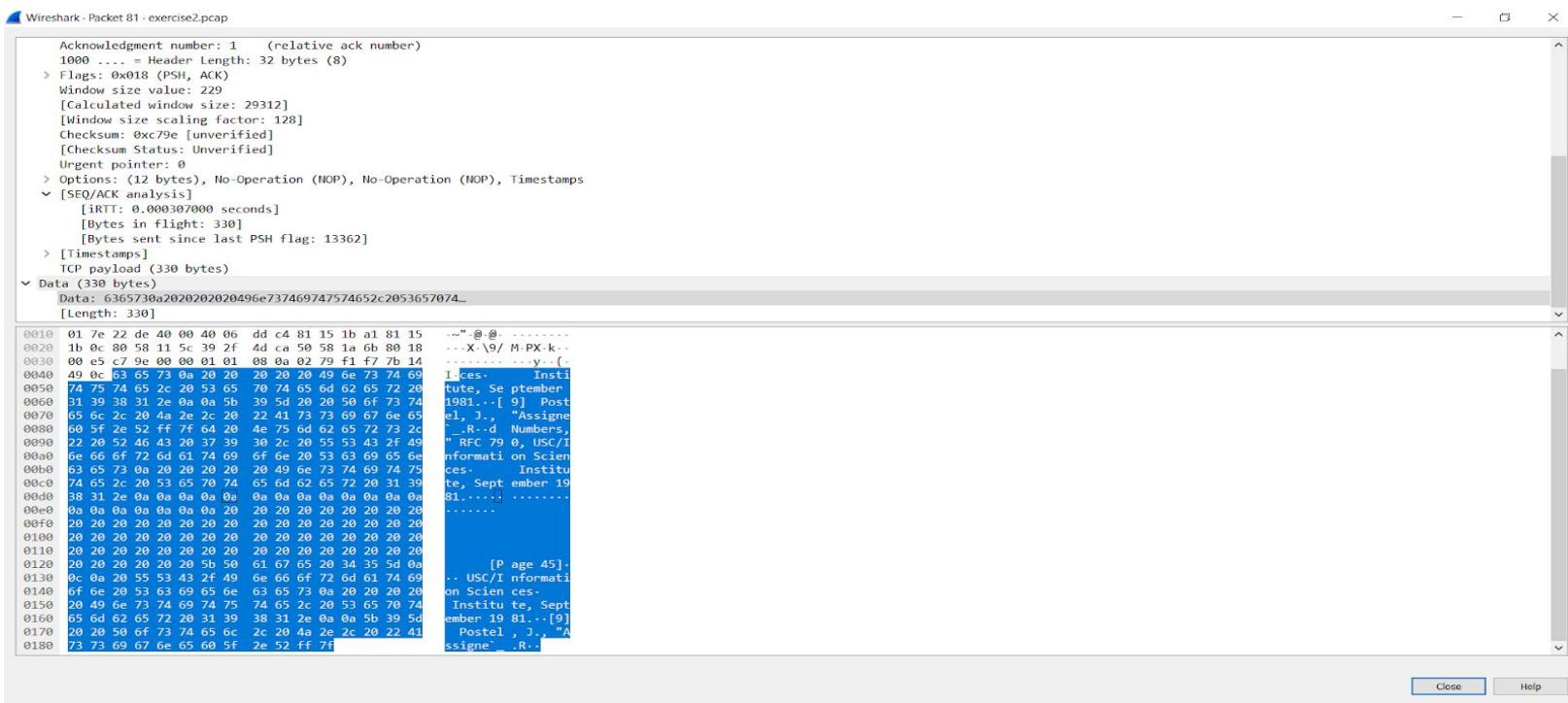


Fig 12: The Last payload sent by the source

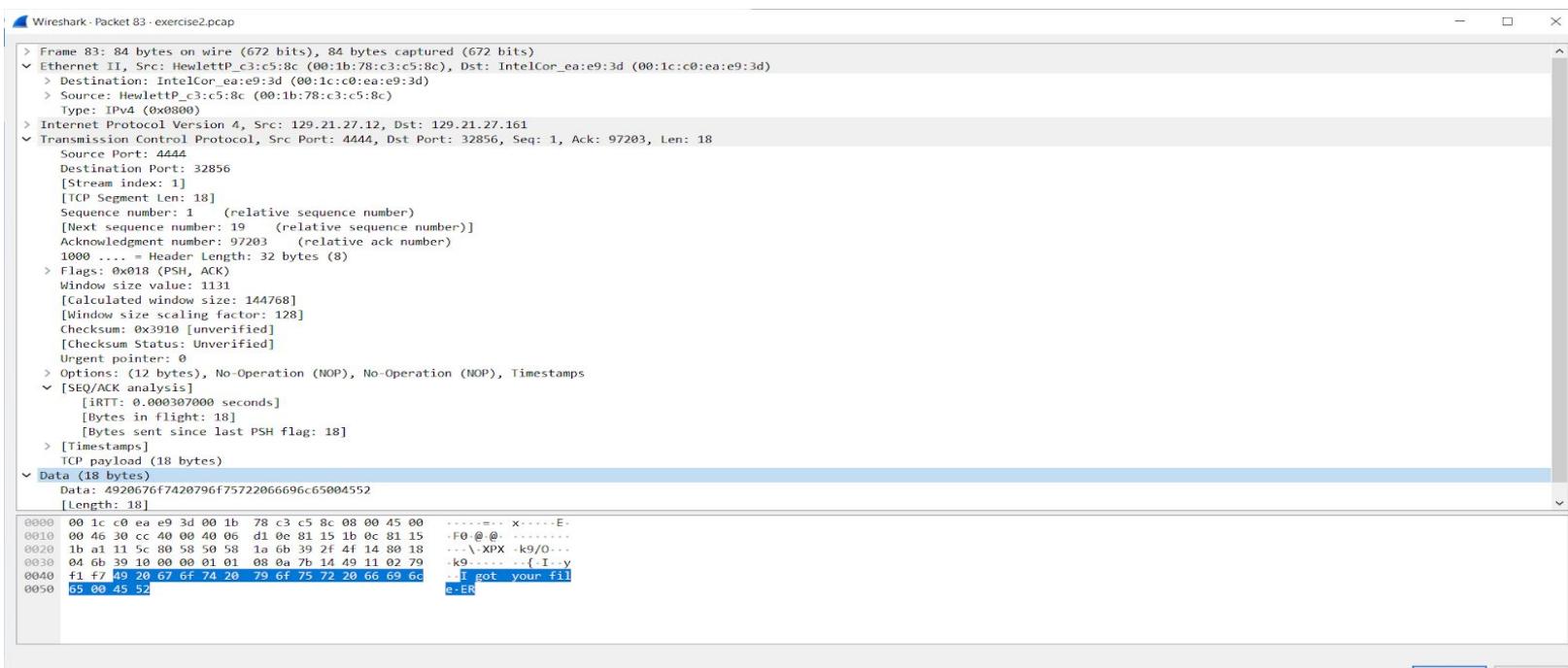


Fig13: Data sent by the destination to the source after completion of the file transfer

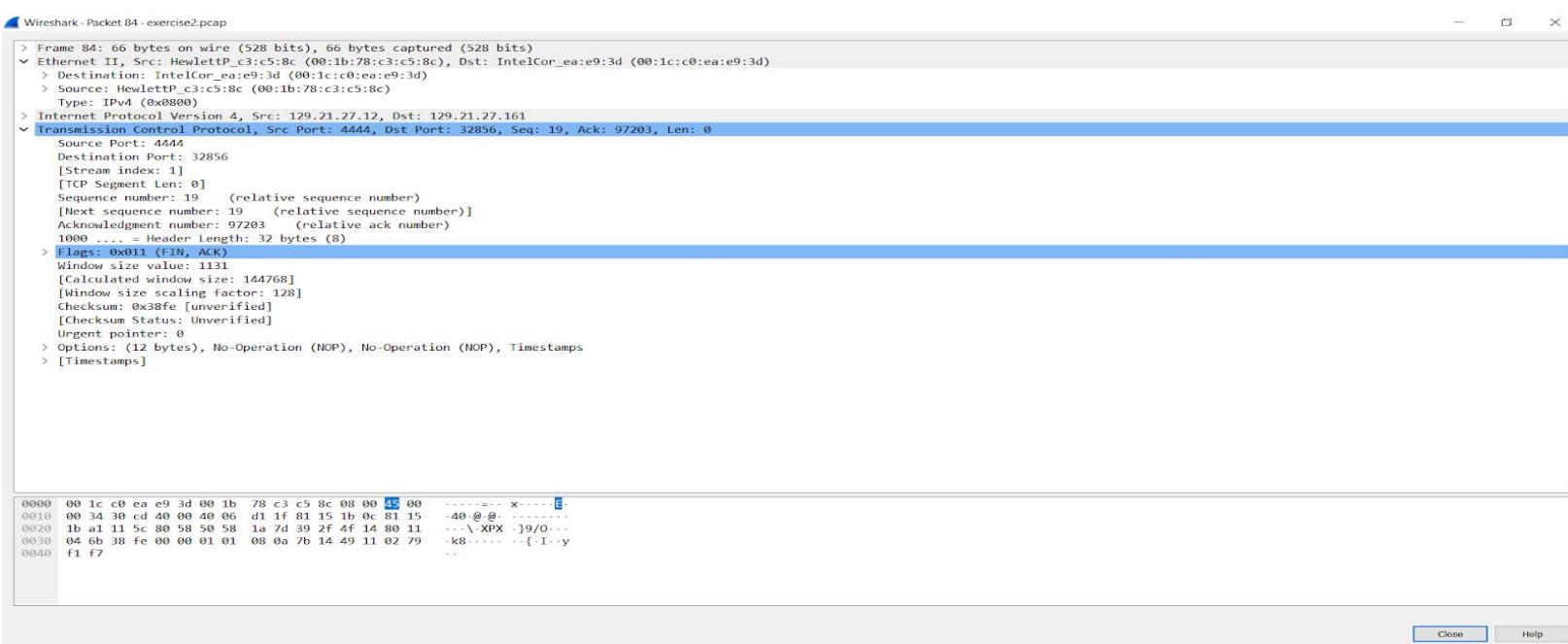


Fig 14: FIN+ACK send by destination to initiate the termination of the TCP connection

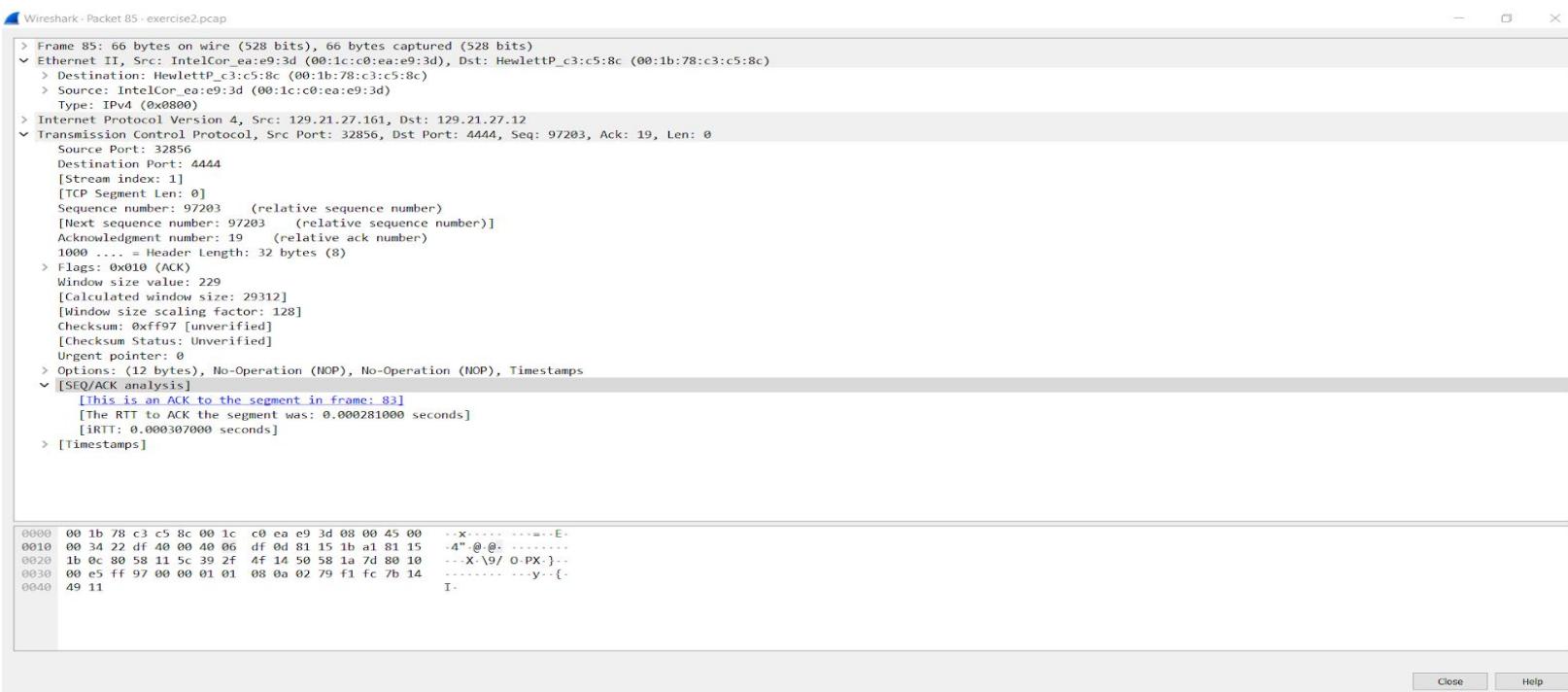


Fig 15: ACK send by the source for the FIN+ACK of destination

Wireshark - Packet 86 - exercise2.pcap

```

> Frame 86: 66 bytes on wire (528 bits), 66 bytes captured (528 bits)
  Ethernet II, Src: IntelCor_ea:e9:3d (00:1c:c0:ea:e9:3d), Dst: HewlettP_c3:c5:8c (00:1b:78:c3:c5:8c)
    > Destination: HewlettP_c3:c5:8c (00:1b:78:c3:c5:8c)
    > Source: IntelCor_ea:e9:3d (00:1c:c0:ea:e9:3d)
    Type: IPv4 (0x0800)
> Internet Protocol Version 4, Src: 129.21.27.161, Dst: 129.21.27.12
  Transmission Control Protocol, Src Port: 32856, Dst Port: 4444, Seq: 97203, Ack: 20, Len: 0
    Source Port: 32856
    Destination Port: 4444
    [Stream index: 1]
    [TCP Segment Len: 0]
    Sequence number: 97203 (relative sequence number)
    [Next sequence number: 97203 (relative sequence number)]
    Acknowledgment number: 20 (relative ack number)
    1000 .... = Header Length: 32 bytes (8)
    > Flags: 0x011 (FIN, ACK)
    Window size value: 229
    [Calculated window size: 29312]
    [Window size scaling factor: 128]
    Checksum: 0xffff [unverified]
    [Checksum Status: Unverified]
    Urgent pointer: 0
    > Options: (12 bytes), No-Operation (NOP), No-Operation (NOP), Timestamps
    < SEQ/ACK analysis
      [This is an ACK to the segment in frame: 84]
      [The RTT to ACK the segment was: 0.000253000 seconds]
      [iRTT: 0.000307000 seconds]
    > [Timestamps]

0000  00 1b 78 c3 c5 8c 00 1c c0 ea e9 3d 08 00 45 00 ..x.... .-.=..E-
0010  00 34 22 e0 40 00 40 06 df 0c 81 15 1b a1 81 15 .4@. @. .....
0020  1b 0c 80 58 11 5c 39 2f 4f 14 50 58 1a 7e 80 11 ...X\9/ O-PX~..
0030  00 e5 ff 95 00 00 01 01 08 0a 02 79 f1 fc 7b 14 .....y...{.
0040  49 11 I.


```

Fig 16: Graceful termination from both the end, FIN+ACK send by the source

Wireshark - Packet 87 - exercise2.pcap

```

> Frame 87: 66 bytes on wire (528 bits), 66 bytes captured (528 bits)
  Ethernet II, Src: HewlettP_c3:c5:8c (00:1b:78:c3:c5:8c), Dst: IntelCor_ea:e9:3d (00:1c:c0:ea:e9:3d)
    > Destination: IntelCor_ea:e9:3d (00:1c:c0:ea:e9:3d)
    > Source: HewlettP_c3:c5:8c (00:1b:78:c3:c5:8c)
    Type: IPv4 (0x0800)
> Internet Protocol Version 4, Src: 129.21.27.12, Dst: 129.21.27.161
  Transmission Control Protocol, Src Port: 4444, Dst Port: 32856, Seq: 20, Ack: 97204, Len: 0
    Source Port: 4444
    Destination Port: 32856
    [Stream index: 1]
    [TCP Segment Len: 0]
    Sequence number: 20 (relative sequence number)
    [Next sequence number: 20 (relative sequence number)]
    Acknowledgment number: 97204 (relative ack number)
    1000 .... = Header Length: 32 bytes (8)
    > Flags: 0x010 (ACK)
    Window size value: 1131
    [Calculated window size: 144768]
    [Window size scaling factor: 128]
    Checksum: 0x38fe [unverified]
    [Checksum Status: Unverified]
    Urgent pointer: 0
    > Options: (12 bytes), No-Operation (NOP), No-Operation (NOP), Timestamps
    < SEQ/ACK analysis
      [This is an ACK to the segment in frame: 86]
      [The RTT to ACK the segment was: 0.000009000 seconds]
      [iRTT: 0.000307000 seconds]
    > [Timestamps]

0000  00 1c c0 ea e9 3d 00 1b 78 c3 c5 8c 08 00 45 00 ..-=..x....E-
0010  00 34 30 ce 40 00 40 06 d1 1e 81 15 1b 0c 81 15 .4@. @. .....
0020  1b a1 11 5c 80 58 50 58 1a 7e 39 2f 4f 15 80 10 ...X\9/ O-PX~..
0030  04 6b 38 fe 00 00 01 01 08 0a 7b 14 49 11 02 79 .k8.....{.I.y
0040  f1 fc ..


```

Fig 17: ACK sent by the receiver and TCP connection is terminated.

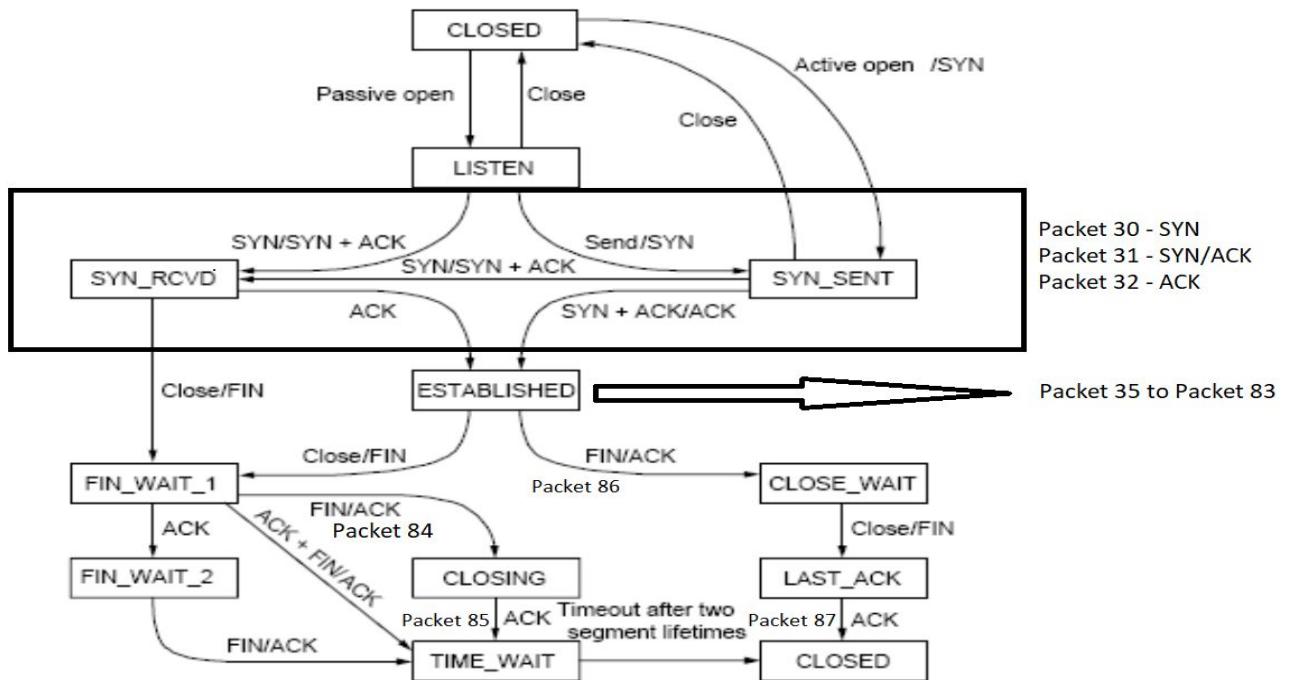


Fig 18: TCP state Diagram

2

b) A structure creates a data type to used group items of possibly different types into a single type. struct sniff_ether is used to store the parameters included in the ethernet frame. It contains two unsigned character arrays of six bytes for source and destination MAC addresses, which are 6 bytes in length. It also contains a two-byte unsigned short variable to get the ether type from the header which specifies whether the frame type is IP, ARP, etc.

"ethernet = (struct sniff_ether*)(pcktbuf); " is typecasting a pointer. In the code, pcktbuf is defined as an unsigned char array of 65535 bytes. By using the above command, the array of unsigned char is typecasted to a struct using a pointer. Typecasting changes the variable type for that operation into a different datatype mentioned in the code.

The other examples of typecasting using a pointer in the same function are,

ip = (struct sniff_ip*)(pcktbuf + SIZE_ETHERNET)

tcp = (struct sniff_tcp*)(pcktbuf + SIZE_ETHERNET + *size_ip)

payload = (u_char*)(pcktbuf + SIZE_ETHERNET + *size_ip + *size_tcp)

"int isgetTCPIP(BYTE *pcktbuf, u_int *size_ip, u_int *size_tcp)" function takes input as a pointer to the pcktbuf which points to the start of the packet buffer, a pointer to IP size and TCP size.

ether = (struct sniff_ether*)(pcktbuf), as discussed before changes the type of variable pcktbuf to a struct of sniff_ether so that the Ethernet frame can be sniffed and information of

MAC addresses and ether type can be obtained from the packet buffer (pcktbuf). After this, the ethernet header is added to the packet buffer and forwarded to the IP layer i.e data link layer. Using ip = (struct sniff_ip*)(pcktbuf + SIZE_ETHERNET); the frame is assigned a type struct sniff_ip so that using the object “ip” of sniff_ip we can access the parameters defined inside that function for our encapsulation process. The size of IP is compared and passed ahead only if it satisfies the requirement of header length. With the help of tcp = (struct sniff_tcp*)(pcktbuf + SIZE_ETHERNET + *size_ip); the data link layer packet is now typecasted using the sniff_tcp structure pointer. The TCP header size is obtained and verified if it is of the correct length. By adding all the headers, namely Ethernet, IP, and TCP we get the location of where data/payload starts i.e. payload = (u_char*)(pcktbuf + SIZE_ETHERNET + *size_ip + *size_tcp);

This function is separating headers from Ethernet protocol, IP protocol and TCP protocol which resides in data link layer, network layer and transport layer respectively.

c)

```
dit Selection Find View Goto Tools Project Preferences Help ~/Desktop/exercise2/skeletoncode.c (Sublime Projects) - Sublime Text (UNREGISTERED)
skeletoncode.c x skeletoncode_q2d.c x skeletoncode_q2e.c x
84
85
86 /* Find stream in file, count packets and get size (in bytes) */
87 if( isgetTCP(pcktbuf, &size_ip, &size_tcp,fp)){
88     /* Simple example code */
89     //fprintf(fp, "packet: %d, Source IPv4 Address: %d, Source Port: %u, Destination IPv4 Address: %d, Destination Port: %u, Sequence Number: %d, Acknowledgement Number: %d\n", pcktcnt,one_s,two_s,three_s,four_s, s_port, d_port, seq_no, ack_no);
90
91     //fprintf(fp, "packet: %d, Source IPv4 Address: %s, Source Port: %u, Destination IPv4 Address: %s, Destination Port: %u, Sequence Number: %d, Acknowledgement Number: %d\n", pcktcnt,one_s,two_s,three_s,four_s, s_port, d_port, seq_no, ack_no);
92
93     BYTE one_s =ip->ip_src.S.un.S.un.b.s.b1;           // Source IPv4 address in standard format of four 8-bit decimal numbers separated by dots
94     BYTE two_s =ip->ip_src.S.un.S.un.b.s.b2;          // Accessing individual 8 bits from inside the struct and union
95     BYTE three_s =ip->ip_src.S.un.S.un.b.s.b3;
96     BYTE four_s =ip->ip_src.S.un.S.un.b.s.b4;
97
98     BYTE one_d =ip->ip_src.S.un.S.un.b.s.b1;           // Destination IPv4 address in standard format of four 8-bit decimal numbers separated by dots
99     BYTE two_d =ip->ip_src.S.un.S.un.b.s.b2;          // Accessing individual 8 bits from inside the struct and union
100    BYTE three_d =ip->ip_src.S.un.S.un.b.s.b3;
101    BYTE four_d =ip->ip_src.S.un.S.un.b.s.b4;
102
103
104    u_short s_port = ush_endian_swp(tcp->th_sport);      // Endian swap for getting network to host (ntoh) format of unsigned short
105    u_short d_port = ush_endian_swp(tcp->th_dport);        // Endian swap for getting network to host (ntoh) format of unsigned short
106
107
108    u_int seq_no = uint_endian_swp(tcp->th_seq);          // Endian swap for getting network to host (ntoh) format of unsigned int
109    u_int ack_no = uint_endian_swp(tcp->th_ack);           // Endian swap for getting network to host (ntoh) format of unsigned int
110
111
112
113    // Print format to get the desired data presentation
114    fprintf(fp, "Packet: %d, Source IPv4 Address: %u.%u.%u.%u, Source Port: %u, ",pcktcnt,one_s,two_s,three_s,four_s, s_port);
115    fprintf(fp, "Destination IPv4 Address: %u.%u.%u.%u, Destination Port: %u, ", one_d,two_d,three_d,four_d, d_port);
116    fprintf(fp, "Sequence Number: %u, Acknowledgement Number: %u\n", seq_no, ack_no);
117
118 } // isgetTCP
119
120 } //while curpos < InLen
121 fclose(InRaw);
122 fclose(fp);
123
124 return(0);
125
126 }
127
128 u_short ush_endian_swp(u_short p)
129 {
130     u_short res;
```

Fig 19: Code to obtain Source IPv4 address and port number, Destination IPv4 address and port number, Sequence number, Acknowledgement number

The encap.h file contains a struct in_addr_new which further contains a union to store the values of IPv4 address in parts as a single-byte i.e 8 bits. I have referenced to those bytes through various structures and stored in different variables which would help in writing them with

a specific format in the file. Similar operations are carried out for destination IPv4 addresses as well.

To overcome the “endianness” problem, source and destination port are endian swapped using the ush_endian_swp function for unsigned short values.

Also, the sequence number and acknowledgment number are endian swapped to get the proper results using `uint_endian_swp` function for unsigned int values.

Sequence numbers and acknowledgment numbers are not relative to each other unlike the output shown in the Wireshark portal in question 1d.

Fig 20: Information for each TCP packet displayed

d)

```
File Edit Selection Find View Goto Tools Project Preferences Help
~/Desktop/exercise2/skeletoncode_q2d.c (Sublime Text (UNREGISTERED))

10 #include <sys/types.h>
11 #include "encap.h"
12 #include <arpa/inet.h>
13 #include <string.h>
14
15
16 u_short ush_endian_swp(u_short p);
17 unsigned int uint_endian_swp(unsigned int p);
18 int isgetTCPIP(BYTE *pcktbuf, u_int *size_ip, u_int *size_tcp,FILE *);
19
20 struct sniff_ethernet *ethernet; /* The ethernet header */
21 struct sniff_tcp *tcp; /* The TCP header */
22 struct sniff_ip *ip; /* The IP header */
23 char *payload; /* Packet payload */
24
25 main(int argc,char **argv)
26 {
27     FILE *InRaw,*fp;
28     struct stat filedat;
29     off64_t InLen, curpos;
30     struct pcap_file_header pcapfilehdr;
31     struct pcap_pkthdr pckthdr;
32     BYTE pcktfif[55535];
33     u_int size_ip;
34     u_int size_tcp;
35     unsigned int pcktcnt=0;
36     int totalsize = 0;
37
38     // Defining Source IPv4 Address in individual bytes
39     int check_ip1 = 129;
40     int check_ip2 = 21;
41     int check_ip3 = 27;
42     int check_ip4 = 161;
43
44     if(argc<2)
45     {
46         printf("ERROR: Too few input arguments\n");
47         printf("Usage: encap input.pcap \n");
48         return 0;
49     }
50     if(stat(*(argv+1),&filedat)==-1)
51     {
52         printf("ERROR: Can't get length of input file\n");
53         return 0;
54     }
55     InLen=filedat.st_size;
56
57 }
```

Fig 21: Source IPv4 defined as four int variables to later compare with the result obtained from code

```
File Edit Selection Find View Goto Tools Project Preferences Help
~/Desktop/exercise2/skeletoncode_q2d.c (Sublime Text (UNREGISTERED))

88     if (fread((char *) &pcktbuf, pckthdr.caplen, 1, InRaw) != 1) {
89         break;
90     }
91
92     /* Find stream in file, count packets and get size (in bytes) */
93     if( isgetTCPIP(pcktbuf, &size_ip, &size_tcp,fp)){
94         /* Simple example code */
95         u_short ip_frame_length = ush_endian_swp(ip->ip_len);
96
97         u_short tcp_packet_length = ip_frame_length - size_ip;
98
99         u_short i_packet_size = tcp_packet_length - size_tcp;
100
101         totalsize += i_packet_size;
102
103         unsigned int seq_no = uint_endian_swp(tcp->th_seq);
104         unsigned int ack_no = uint_endian_swp(tcp->th_ack);
105
106         BYTE one_s = ip->ip_src.S.un.S.un.b.s.b1;           // Source IPv4 address in standard format of four 8-bit decimal numbers separated by dots
107         BYTE two_s = ip->ip_src.S.un.S.un.b.s.b2;           // Accessing individual 8 bits from inside the struct and union
108         BYTE three_s = ip->ip_src.S.un.S.un.b.s.b3;
109         BYTE four_s = ip->ip_src.S.un.S.un.b.s.b4;
110
111
112         // Check if payload is actually containing data or not
113         if (i_packet_size != 0)
114         {
115             // Last data sent by Receiver is not the fragment of file transmission thus discarded. Only transmission from Source is considered
116             if (check_ip1==one_s && check_ip2==two_s && check_ip3==three_s && check_ip4==four_s) |
117             {
118                 fprintf(fp, "Packet no.: %d, Size of Payload is %u bytes, Sequence Number: %u\n",pcktcnt, i_packet_size, seq_no);
119             }
120         }
121
122     }
123
124 } // isgetTCPIP
125
126 } //while currpos < InLen
127
128 fprintf(fp, "Total file size is %d bytes\n",totalsize);
129 fclose(InRaw);
130 fclose(fp);
131
132 return(0);
133
134 }
```

Fig 22: Operations carried out to find each packet data size

Firefox Web Browser

outdata_q2d.txt
~/Desktop/exercise2

```

Packet no.: 35, Size of Payload is 262 bytes, Sequence Number: 959304546
Packet no.: 37, Size of Payload is 1448 bytes, Sequence Number: 959304808
Packet no.: 39, Size of Payload is 1448 bytes, Sequence Number: 959306256
Packet no.: 41, Size of Payload is 5792 bytes, Sequence Number: 959307704
Packet no.: 43, Size of Payload is 5792 bytes, Sequence Number: 959313496
Packet no.: 45, Size of Payload is 1764 bytes, Sequence Number: 959319288
Packet no.: 47, Size of Payload is 5792 bytes, Sequence Number: 959321052
Packet no.: 49, Size of Payload is 4344 bytes, Sequence Number: 959326844
Packet no.: 51, Size of Payload is 4344 bytes, Sequence Number: 959331188
Packet no.: 53, Size of Payload is 4908 bytes, Sequence Number: 959335532
Packet no.: 55, Size of Payload is 2896 bytes, Sequence Number: 959349440
Packet no.: 57, Size of Payload is 4344 bytes, Sequence Number: 959343336
Packet no.: 59, Size of Payload is 2896 bytes, Sequence Number: 959347680
Packet no.: 61, Size of Payload is 4344 bytes, Sequence Number: 959350576
Packet no.: 63, Size of Payload is 7240 bytes, Sequence Number: 959354920
Packet no.: 65, Size of Payload is 5792 bytes, Sequence Number: 959362160
Packet no.: 67, Size of Payload is 5792 bytes, Sequence Number: 959367952
Packet no.: 69, Size of Payload is 5792 bytes, Sequence Number: 959373744
Packet no.: 71, Size of Payload is 5792 bytes, Sequence Number: 959379536
Packet no.: 73, Size of Payload is 3058 bytes, Sequence Number: 959385328
Packet no.: 75, Size of Payload is 4344 bytes, Sequence Number: 959388386
Packet no.: 77, Size of Payload is 4344 bytes, Sequence Number: 959392730
Packet no.: 79, Size of Payload is 4344 bytes, Sequence Number: 959397074
Packet no.: 81, Size of Payload is 330 bytes, Sequence Number: 959401418
Total file size is 97220 bytes

```

Fig 23: Output for question 2d

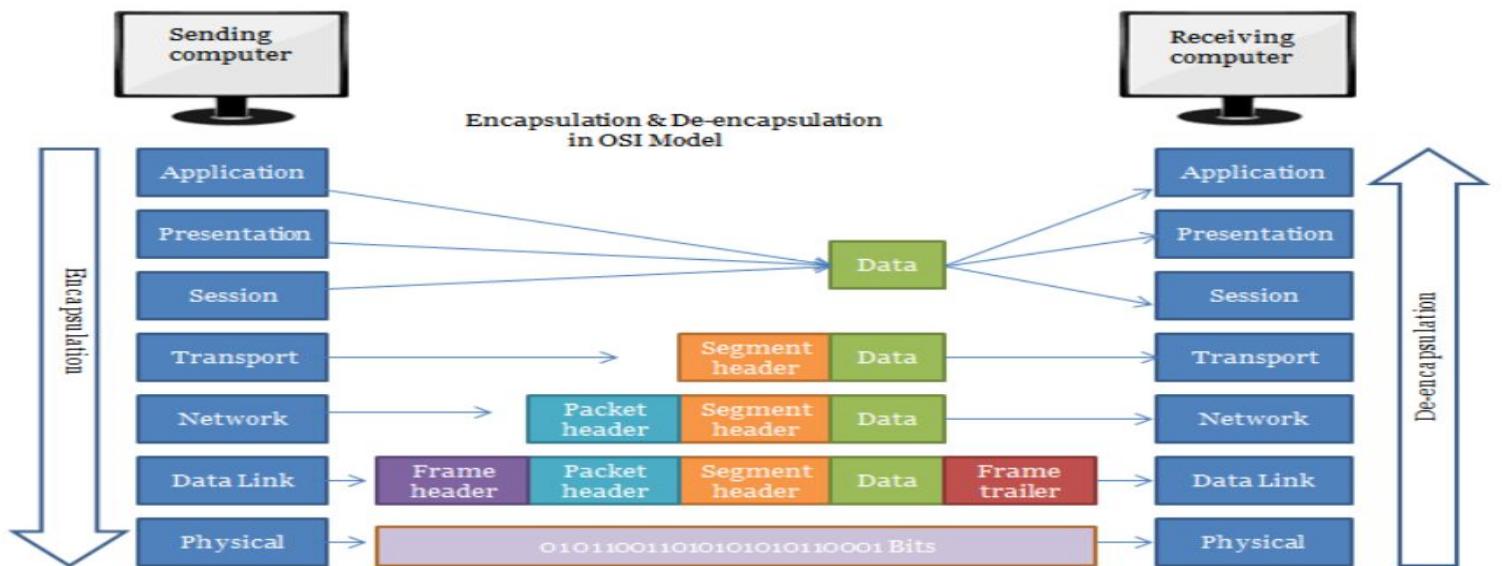


Fig 24: Encapsulation & De-encapsulation in OSI model

We know that different layers add their header to the message coming from the upper layers. For accessing the data, these header files must be removed. In Fig 22, the IP header is

removed from the IP segment. The resultant message still contains the TCP header which is removed to obtain the payload (Data) and its size.

We need to get only those packets that contain the fragment of file data transfer. Packet 83 sends an acknowledgment data from the receiver to the source which is not the data of the file transferred. Hence this packet is discarded here with the use of source IPv4 address comparison.

For getting the un-relative sequence and acknowledgment number, Edit -> Preference -> Protocols -> TCP -> Turn off the relative sequence number.

exercise2.pcap

File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help

Apply a display filter ... <Ctrl-/>

Expression...

No.	Time	Source	Destination	Protocol	Length	Info
34	3.212339	IntelCor_ea:e9:3d	Hewlett_Pc:c5:8c	ARP	60	129.21.27.161 is at 00:1c:00:ea:e9:3d
35	3.212347	129.21.27.161	129.21.27.12	TCP	328	32856 → 4444 [PSH, ACK] Seq=959304546 Ack=1347951211 Win=29312 Len=262 TSval=41546231 TSecr=2064926988
36	3.212357	129.21.27.12	129.21.27.161	TCP	66	4444 → 32856 [ACK] Seq=1347951211 Ack=959304808 Win=30080 Len=0 TSval=2064926988 TSecr=41546231
37	3.212361	129.21.27.161	129.21.27.12	TCP	1514	32856 → 4444 [ACK] Seq=9593048088 Ack=1347951211 Win=29312 Len=144 TSval=41546231 TSecr=2064926988
38	3.212368	129.21.27.12	129.21.27.161	TCP	66	4444 → 32856 [ACK] Seq=1347951211 Ack=959306256 Win=33024 Len=0 TSval=2064926988 TSecr=41546231
39	3.212389	129.21.27.161	129.21.27.12	TCP	1514	32856 → 4444 [ACK] Seq=1347951211 Ack=9593062568 Win=29312 Len=144 TSval=41546231 TSecr=2064926988
40	3.212394	129.21.27.12	129.21.27.161	TCP	66	4444 → 32856 [ACK] Seq=1347951211 Ack=959307704 Win=35840 Len=0 TSval=2064926988 TSecr=41546231
41	3.212445	129.21.27.161	129.21.27.12	TCP	5858	32856 → 4444 [ACK] Seq=959307704 Ack=1347951211 Win=29312 Len=5792 TSval=41546231 TSecr=2064926988
42	3.212451	129.21.27.12	129.21.27.161	TCP	66	4444 → 32856 [ACK] Seq=1347951211 Ack=959313496 Win=47488 Len=0 TSval=2064926988 TSecr=41546231
43	3.212495	129.21.27.161	129.21.27.12	TCP	5858	32856 → 4444 [ACK] Seq=959313496 Ack=1347951211 Win=29312 Len=5792 TSval=41546231 TSecr=2064926988
44	3.212501	129.21.27.12	129.21.27.161	TCP	66	4444 → 32856 [ACK] Seq=1347951211 Ack=959319288 Win=59008 Len=0 TSval=2064926988 TSecr=41546231
45	3.212538	129.21.27.161	129.21.27.12	TCP	1830	32856 → 4444 [PSH, ACK] Seq=959319288 Ack=1347951211 Win=29312 Len=1764 TSval=41546231 TSecr=2064926988
46	3.212544	129.21.27.12	129.21.27.161	TCP	66	4444 → 32856 [ACK] Seq=1347951211 Ack=959321052 Win=62592 Len=0 TSval=2064926988 TSecr=41546231
47	3.212559	129.21.27.161	129.21.27.12	TCP	5858	32856 → 4444 [ACK] Seq=959321052 Ack=1347951211 Win=29312 Len=5792 TSval=41546231 TSecr=2064926988
48	3.212565	129.21.27.12	129.21.27.161	TCP	66	4444 → 32856 [ACK] Seq=1347951211 Ack=959326844 Win=74112 Len=0 TSval=2064926988 TSecr=41546231
49	3.212591	129.21.27.161	129.21.27.12	TCP	4410	32856 → 4444 [ACK] Seq=959326844 Ack=1347951211 Win=29312 Len=4340 TSval=41546231 TSecr=2064926988
50	3.212597	129.21.27.12	129.21.27.161	TCP	66	4444 → 32856 [ACK] Seq=1347951211 Ack=959331188 Win=82816 Len=0 TSval=2064926988 TSecr=41546231
51	3.212650	129.21.27.161	129.21.27.12	TCP	4410	32856 → 4444 [ACK] Seq=959331188 Ack=1347951211 Win=29312 Len=4340 TSval=41546231 TSecr=2064926988
52	3.212656	129.21.27.12	129.21.27.161	TCP	66	4444 → 32856 [ACK] Seq=1347951211 Ack=959335532 Win=91520 Len=0 TSval=2064926988 TSecr=41546231
53	3.212698	129.21.27.161	129.21.27.12	TCP	4974	32856 → 4444 [PSH, ACK] Seq=959335532 Ack=1347951211 Win=4908 TSval=41546231 TSecr=2064926988
54	3.212705	129.21.27.12	129.21.27.161	TCP	66	4444 → 32856 [ACK] Seq=1347951211 Ack=959340440 Win=101376 Len=0 TSval=2064926988 TSecr=41546231
55	3.212713	129.21.27.161	129.21.27.12	TCP	2962	32856 → 4444 [ACK] Seq=959340440 Ack=1347951211 Win=29312 Len=2896 TSval=41546231 TSecr=2064926988

[Bytes sent since last PSH flag: 8688]
TCP payload (4344 bytes)

Selected Data (4344 bytes):
Data: 2020202020203320202020202020202020202020202020203420...

Hex dump:

0040	49 0c 20 20 20 20 20 20 33 20 20 20 20 20 20 20	1	3
0050	7c 20 20 20 20 20 20 20 20 34 20 20 20 20 20 20 20	4	
0060	7c 0a 20 20 20 20 2b 2d 2b 2d 2b 2d 2b 2d 2b 2d 2b	-	++-----+
0070	2d 2b 2d	-	+++++ + + + +
0080	2d 2b 2b	-	+++++ + + + +
0090	2d 2b 2b	-	+++++ + + + +
00a0	2d 2b 2d 2b 2d 2b 2d 2b 0a 20 20 60 5f 2e 52 ff 7f	-	-----R-----
00b0	7c 20 20 20 20 20 20 20 35 20 20 20 20 20 20 20 20	5	
00c0	7c 20 20 20 20 20 20 20 36 20 20 20 20 20 20 20 20	6	
00d0	7c 20 20 20 20 20 20 20 37 20 20 20 20 20 20 20 20	7	
00e0	7c 20 20 20 20 20 20 20 38 20 20 20 20 20 20 20 20	8	
00f0	7c 0a 20 20 20 20 2b 2d 2b 2d 2b 2d 2b 2d 2b 2d 2b	-	++-----+
0100	2d 2b 2b	-	+++++ + + + +

Data (data.data), 4344 bytes

Packets: 88 • Displayed: 88 (100.0%)

Profile: Default

Fig 25: Updating the sequence and acknowledgment number

We can also check if our obtained packet sizes are correct by using the sequence numbers. We know that the sequence number helps the TCP on both sides keep track of how much data has been transferred and to put the data back into the correct order if it is received in the wrong order, and to request data when it has been lost in transit. It is the first byte indexed in that packet.

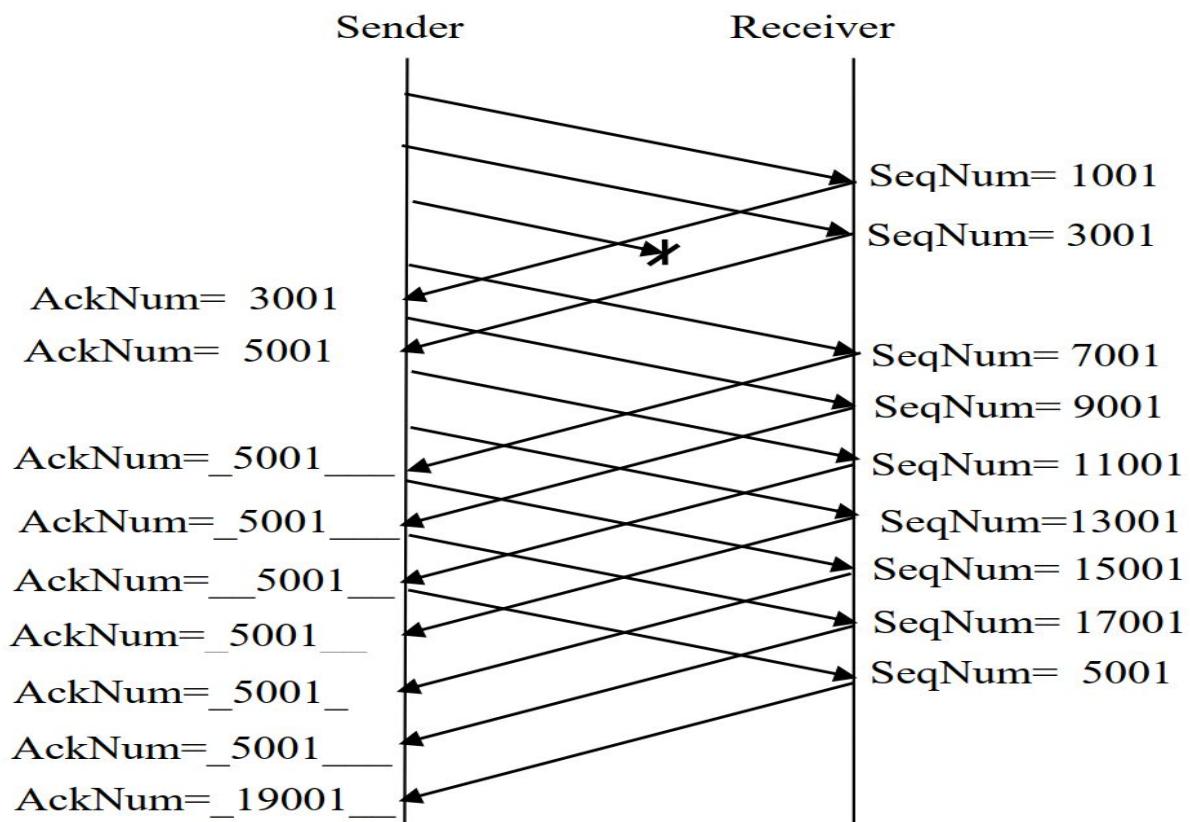


Fig 26: The data transfer in the first packets would be $3001-1001 = 2000$ bytes.

Therefore consider packets 37 and 39 which are transmitted by source and include data in it.

Sequence number_P37 = 959304808

Sequence number_P39 = 959306256

Data transferred in packet 37 = $959306256 - 959304808 = 1448$ bytes

So, we can confirm that the packet data size obtained from the code is correct. Similarly, we can also check the Wireshark portal to confirm this result.

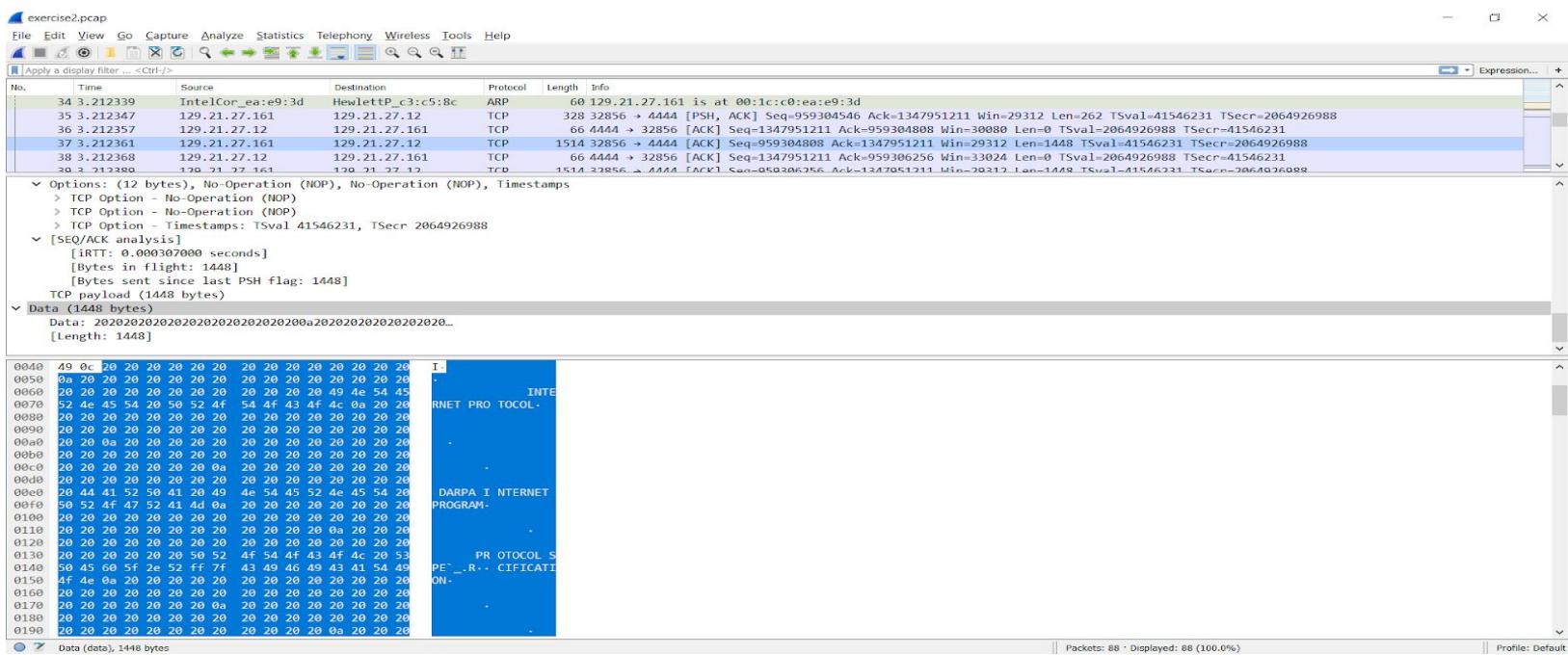


Fig 27: Packet 37 has a payload of 1448 bytes

The total file size is 97.22 kilobytes

e)

```

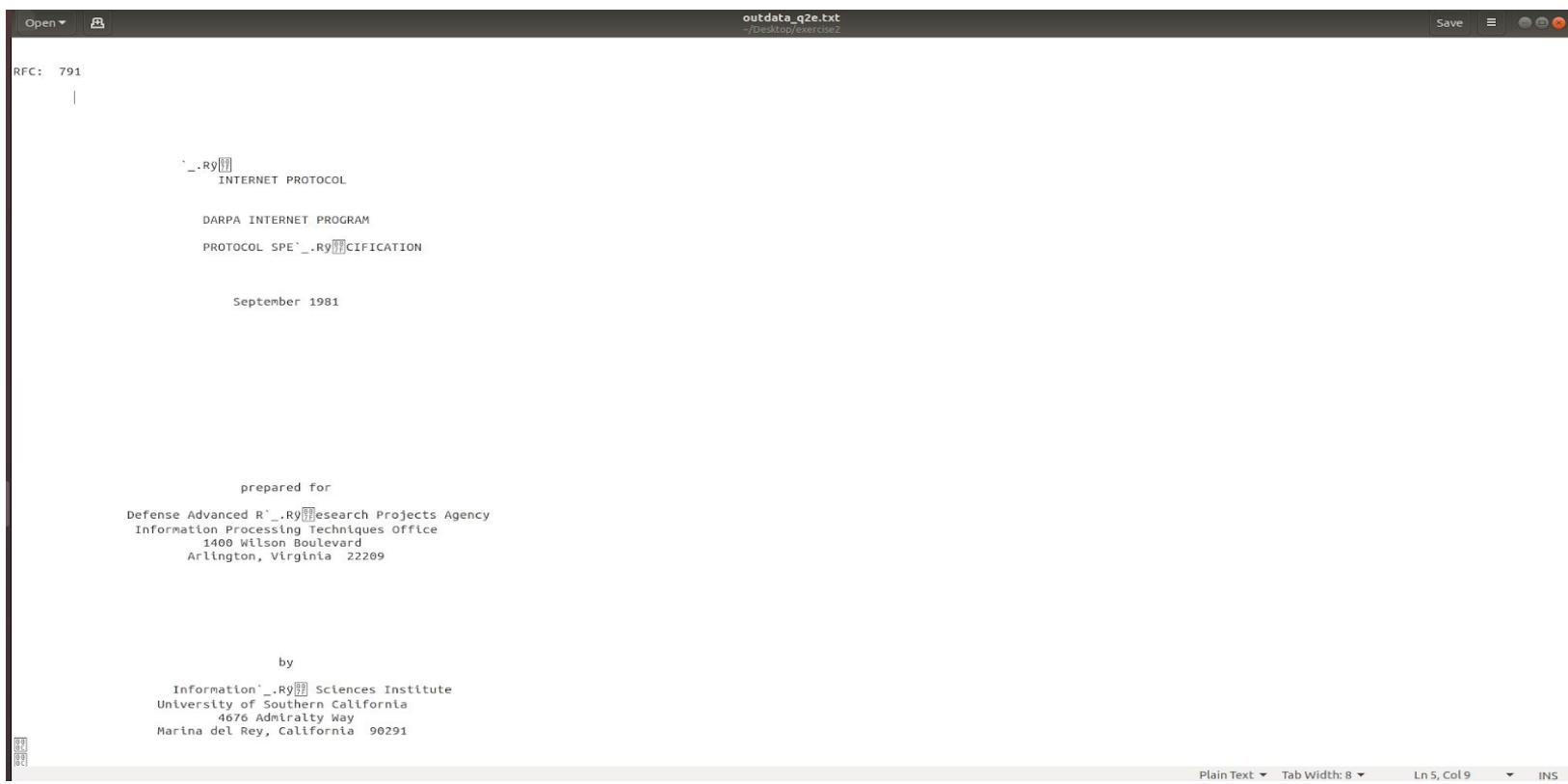
File Edit Selection Find View Goto Tools Project Preferences Help
-/Desktop/exercise2/skeletoncode_q2e.c (Sublime Text (UNREGISTERED))

FOLD skeletoncode.c x skeletoncode_q2d.c x skeletoncode_q2e.c x
84     if ((read((char *) &pckthdr, sizeof(pckthdr), 1, InRaw) != 1) {
85         break;
86     }
87
88     if ((read((char *) &pcktbuff, pckthdr.caplen, 1, InRaw) != 1) {
89         break;
90     }
91
92     /* Find stream in file, count packets and get size (in bytes) */
93     if( isgetTCP(pcktbuff, &size_ip, &size_tcp,fp)){
94         /* Simple example code */
95         u_short ip_frame_length = ush_endian_swp(ip->ip_len);
96
97         u_short tcp_packet_length = ip_frame_length - size_ip;
98
99         u_short i_packet_size = tcp_packet_length - size_tcp;
100
101         // totalsize += i_packet_size;
102
103         unsigned int seq_no = uint_endian_swp(tcp->th_seq);
104         unsigned int ack_no = uint_endian_swp(tcp->th_ack);
105
106         BYTE one_s = ip->ip_src.S.un.S.un.b.s_b1;           // Source IPv4 address in standard format of four 8-bit decimal numbers separated by dots
107         BYTE two_s = ip->ip_src.S.un.S.un.b.s_b2;          // Accessing individual 8 bits from inside the struct and union
108         BYTE three_s = ip->ip_src.S.un.S.un.b.s_b3;
109         BYTE four_s = ip->ip_src.S.un.S.un.b.s_b4;
110
111         if (i_packet_size != 0)
112         {
113             // Last data sent by Receiver is not the fragment of file transmission thus discarded. Only transmission from Source is considered
114             if (check_ip1==one_s && check_ip2==two_s && check_ip3==three_s && check_ip4==four_s)
115             {
116                 int count = 0;
117                 while(count < i_packet_size)
118                 {
119                     fprintf(fp, "%c", *(payload + count));      //Printing into the file byte by byte.
120                     count++;
121                 }
122             }
123         }
124
125         //fprintf(fp, "%s",payload);                         //Using this, it prints all the padding and garbage data along with file content
126     } // isgetTCP();
127
128 } //while currpos < InLen
129
130 fclose(InRaw);

```

Fig 28: Writing the file data used for transfer in the pcap file

Packet 83 sends an acknowledgment data from the receiver to the source which is not the data of the file transferred. Hence this packet is discarded here with the use of source IPv4 address comparison. I have used byte to byte data write into the file using the count variable.



The screenshot shows a text editor window with the following content:

```
RFC: 791
|  
  
`_Ry INTERN PROTOCOL  
INTERNET PROGRAM  
PROTOCOL SPE`_RyIFICATION  
  
September 1981  
  
prepared for  
Defense Advanced R`_Ry Research Projects Agency  
Information Processing Techniques Office  
1400 Wilson Boulevard  
Arlington, Virginia 22209  
  
by  
Information`_Ry Sciences Institute  
University of Southern California  
4676 Admiralty Way  
Marina del Rey, California 90291
```

File path: outdata_q2e.txt
Save | Open ▾

Fig 29: Text file used in the TCP packet transfer

The subject of the file is RFC 791, Internet Protocol, DARPA Internet Protocol Specifications.

References:

- https://www.wireshark.org/docs/wsug_html_chunked/ChCustPreferencesSection.html
- <https://www.computernetworkingnotes.com/ccna-study-guide/data-encapsulation-and-de-encapsulation-explained.html>