ECE F343: Communication Networks
Assignment 1

Pranav Chandra N. V.
2023AAPS0013P

26/06/2026

Contents

Notes

All files and additional source files may be found on the associated GitHub repository. All programs were written/run on a Ubuntu system.

1 Quesition 1

1.1 Introduction

This program was written for Assignment 1, Question 1. The goal is to show how message encapsulation works in the OSI model by simulating the downward flow of data from Layer 7 (Application) to Layer 1 (Physical).

The user enters a message of up to 80 characters. The program passes it through seven functions, one per layer. Each function sticks its own header onto the front of whatever it received, prints the result, and then calls the function for the layer below it.

1.2 OSI Reference Model Overview

The OSI model splits network communication into seven layers, each with a specific job. Table 1 lists each layer, the name for its unit of data (PDU), and the header string used in this program.

Layer	Name	PDU	Simulated Header
7	Application	Message	`[APP: HTTP GET /index.html]`
6	Presentation	Message	`[PRES: ENC=UTF-8 FORMAT=ASCII COMPRESS=NONE]`
5	Session	Message	`[SESS: ID=A1B2C3 SEQ=001 TYPE=DATA]`
4	Transport	Segment	`[TRAN: TCP SRC=1234 DST=80 SEQ=100 ACK=0]`
3	Network	Packet	`[NET: SRC=192.168.1.1 DST=10.0.0.1 TTL=64]`
2	Data Link	Frame	`[DL: SRC=AA:BB:CC DST=11:22:33 TYPE=IPv4]`
1	Physical	Bits	`[PHY: ENC=NRZ SIGNAL=DIGITAL]`

Table 1: OSI Layers, PDU names, and simulated headers

1.3 Design Decisions

- A helper function `prepend_header()` takes care of copying the header and message into a new buffer, so the layer functions themselves stay short and readable.
- Buffers are fixed-size arrays on the stack (`MAX_BUF = 1024` bytes). This is more than enough for seven headers plus an 80-char message, and the helper exits if that ever overflows.
- All headers are kept under 64 characters as required by the spec.
- Each layer function just calls the one below it directly, keeping the call chain simple and easy to follow.

1.4 Constants and Macros

```c
#define MAX_APP_MSG   80
#define MAX_HEADER    64
#define MAX_BUF       1024

#define HDR_APP    "[APP: HTTP GET /index.html]"
#define HDR_PRES   "[PRES: ENC=UTF-8 FORMAT=ASCII COMPRESS=NONE]"
#define HDR_SESS   "[SESS: ID=A1B2C3 SEQ=001 TYPE=DATA]"
#define HDR_TRAN   "[TRAN: TCP SRC=1234 DST=80 SEQ=100 ACK=0]"
#define HDR_NET    "[NET: SRC=192.168.1.1 DST=10.0.0.1 TTL=64]"
#define HDR_DL     "[DL: SRC=AA:BB:CC DST=11:22:33 TYPE=IPv4]"
#define HDR_PHY    "[PHY: ENC=NRZ SIGNAL=DIGITAL]"
```

Listing 1: Buffer and header constants

1.5 Helper: `prepend_header()`

```c
static int prepend_header(const char *header, const char *msg,
   int msg_size, char *out_buf, int out_buf_size){
    int hdr_len  = (int)strlen(header);
    int new_size = hdr_len + msg_size;

    if (new_size >= out_buf_size) {
        fprintf(stderr, "Buffer overflow prevented.\n");
        exit(1);
    }

    memcpy(out_buf,            header, hdr_len);
    memcpy(out_buf + hdr_len, msg,    msg_size);
    out_buf[new_size] = '\0';

    return new_size;
}
```

Listing 2: Header prepending helper

1.6 Layer Functions

All seven layer functions work the same way:

1. Allocate a local buffer (`MAX_BUF` bytes).
2. Call `prepend_header()` to build the new PDU.
3. Print it.
4. Call the next layer down (the Physical layer skips this last step).

```c
void application_layer(char *msg, int size)
{
    char buf[MAX_BUF];
    int  new_size = prepend_header(HDR_APP, msg, size, buf,
       MAX_BUF);
    printf("[Layer 7 - Application]  PDU: %s\n\n", buf);
    presentation_layer(buf, new_size);
}
```

Listing 3: Application Layer (representative example)

1.7 Message Flow Flowchart

Figure 1 shows the message moving down through the layers. The PDU gets longer at each step as another header is added to the front.

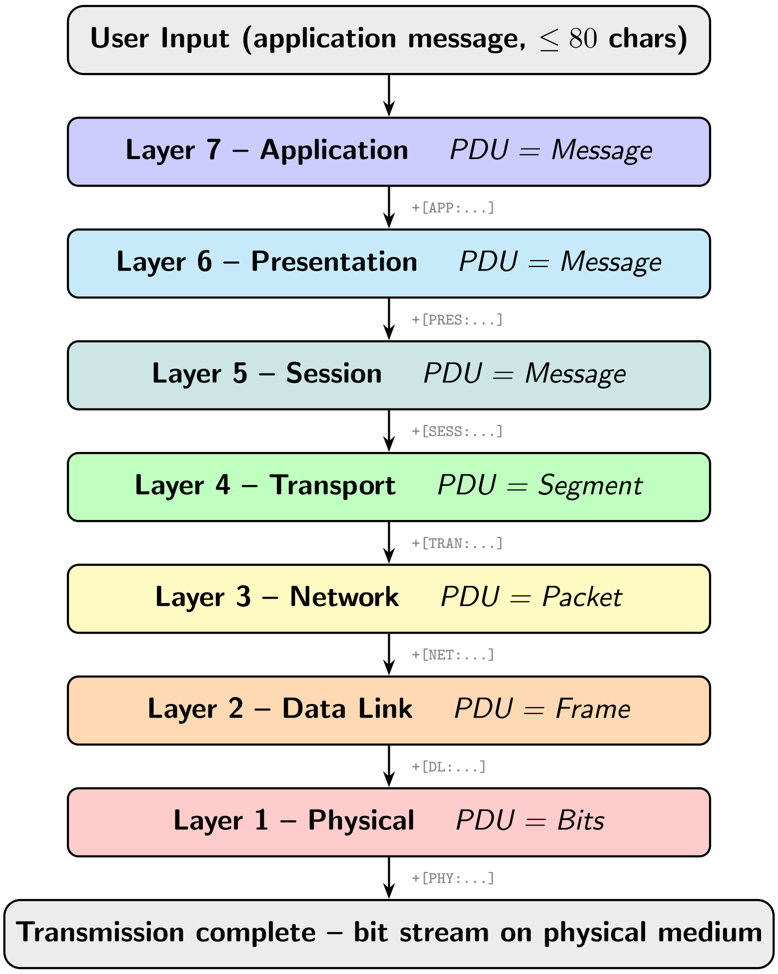

Figure 1: Message flow and encapsulation across the OSI 7 layers.

1.8 Sample Output

Running the program with the input `"Hello, Network!"` gives:

```
Enter application message (up to 80 characters):
> hello world!

=== OSI Encapsulation ===

[Layer 7 - Application]  PDU:
[APP: HTTP GET /index.html]hello world!

[Layer 6 - Presentation] PDU:
[PRES: ENC=UTF-8 FORMAT=ASCII COMPRESS=NONE][APP: HTTP GET /index.html]hello world!

[Layer 5 - Session]      PDU:
[SESS: ID=A1B2C3 SEQ=001 TYPE=DATA][PRES: ENC=UTF-8 FORMAT=ASCII COMPRESS=NONE][APP: HTTP GET /index.html]hello world!

[Layer 4 - Transport]    Segment:
[TRAN: TCP SRC=1234 DST=80 SEQ=100 ACK=0][SESS: ID=A1B2C3 SEQ=001 TYPE=DATA][PRES: ENC=UTF-8 FORMAT=ASCII COMPRESS=NONE][APP: HTTP GET /index.html]hello world!

[Layer 3 - Network]      Packet:
[NET: SRC=192.168.1.1 DST=10.0.0.1 TTL=64][TRAN: TCP SRC=1234 DST=80 SEQ=100 ACK=0][SESS: ID=A1B2C3 SEQ=001 TYPE=DATA][PRES: ENC=UTF-8 FORMAT=ASCII COMPRESS=NONE][APP: HTTP GET /index.html]hello world!

[Layer 2 - Data Link]    Frame:
[DL: SRC=AA:BB:CC DST=11:22:33 TYPE=IPv4][NET: SRC=192.168.1.1 DST=10.0.0.1 TTL=64][TRAN: TCP SRC=1234 DST=80 SEQ=100 ACK=0][SESS: ID=A1B2C3 SEQ=001 TYPE=DATA][PRES: ENC=UTF-8 FORMAT=ASCII COMPRESS=NONE][APP: HTTP GET /index.html]hello world![FCS: 0xF47B14F0]
                         FCS (CRC-32): 0xF47B14F0

[Layer 1 - Physical]     Bits:
[PHY: ENC=NRZ SIGNAL=DIGITAL][DL: SRC=AA:BB:CC DST=11:22:33 TYPE=IPv4][NET: SRC=192.168.1.1 DST=10.0.0.1 TTL=64][TRAN: TCP SRC=1234 DST=80 SEQ=100 ACK=0][SESS: ID=A1B2C3 SEQ=001 TYPE=DATA][PRES: ENC=UTF-8 FORMAT=ASCII COMPRESS=NONE][APP: HTTP GET /index.html]hello world![FCS: 0xF47B14F0]

=== Transmission complete: 288 bytes on the wire ===
```

Listing 4: Sample program output

1.9 Build and Run

The project uses a simple `Makefile`:

```
CC     = gcc
CFLAGS = -Wall -Wextra -std=c11
TARGET = osi_model
SRC    = osi_model.c

all: $(TARGET)

$(TARGET): $(SRC)
	$(CC) $(CFLAGS) -o $(TARGET) $(SRC)

clean:
	rm -f $(TARGET)
```

Listing 5: Makefile

Compile and run with:

```
make
./osi_model
```

1.10 Summary

This program gives a rough idea of how encapsulation works going down the OSI stack. Each layer adds its own header before passing the data on, so by the time it reaches Layer 1 the original message is buried inside several headers. The headers here are simplified — they do not carry real protocol data — but the structure matches what each layer is actually responsible for: application requests at Layer 7, encoding info at Layer 6, session tracking at Layer 5, port and sequence numbers at Layer 4, IP addresses at Layer 3, MAC addresses at Layer 2, and signalling info at Layer 1.

Table 2 shows the cumulative PDU size at each layer for the sample input `"Hello, Network!"` (15 bytes).

Layer	Name	Header/Trailer (bytes)	Cumulative PDU (bytes)
–	User Input	15 (message)	15
7	Application	27	42
6	Presentation	44	86
5	Session	35	121
4	Transport	41	162
3	Network	42	204
2	Data Link	41 + 17 (FCS)	262
1	Physical	29	291

Table 2: Byte count at each OSI layer (sample message: `"Hello, Network!"`)

2 Question 2

2.1 Wireshark Protocols

Question: What protocols are listed in the Wireshark "protocol" column in your trace file? Make a list of such protocols, identify the layer to which they belong, and briefly explain (in 1-2 lines) the function of each protocol.

Answer: The protocols listed under the "protocol" column are shown below, grouped by their OSI layer.

2.1.1 Layer 2 – Data Link

- **LLC** – Identifies upper-layer protocols within Ethernet frames.
- **ARP** – Maps an IPv4 address to a MAC address on a local network.

2.1.2 Layer 3 – Network

- **IGMPv2** – Manages IPv4 multicast group membership.
- **IGMPv3** – Supports source-specific IPv4 multicast membership.
- **ICMPv6** – Provides IPv6 error reporting and neighbor discovery.
- **VRRP** – Provides gateway redundancy using a virtual IP address.

2.1.3 Layer 4 – Transport

- **TCP** – Provides reliable, connection-oriented data transmission.
- **UDP** – Provides fast, connectionless data transmission.

2.1.4 Layer 6/7 – Presentation / Application

- **TLSv1.2** – Encrypts and secures application data.
- **TLSv1.3** – Provides faster and more secure encryption than TLS 1.2.

2.1.5 Layer 7 – Application

- **mDNS** – Resolves hostnames locally using multicast without a DNS server.
- **SSDP** – Discovers devices and services on a local network.
- **DHCP** – Automatically assigns IP configuration to clients.
- **DNS** – Translates domain names into IP addresses.
- **NBNS** – Resolves NetBIOS names to IP addresses.
- **HTTP** – Transfers web content using a request-response model.
- **BROWSER** – Maintains shared resource lists in Windows networks.

2.2 HTTP Protocl Stats

Question: Read about HTTTP protocol and its working (Reference: Section 2.1, 8.1 in Garcia). Now in your experiment, determine how long did it take from when the HTTP GET message was sent until the HTTP OK reply was received? (By default, the value of the Time column in the packet-listing window is the amount of time, in seconds, since Wireshark tracing began. (If you want to display the Time field in time-of-day format, select the Wireshark View pull down menu, then select Time Display Format, then select Time-of-day.)

Answer: The timing of the HTTP GET/OK exchange is summarised in Table 3.

Event	**Timestamp (s)**
HTTPS GET sent	19.084048510
HTTP OK received	19.830304061
Round-trip time	**0.746255551 s**

Table 3: HTTP GET / OK Timing

Note: I used another HTTP website (this one) to test HTTP, as I couldn't get the gaia website working even after turning of HTTPS3.

2.3 Internet Address of Gaia

Question What is the Internet address of the gaia.cs.umass.edu (also known as www-net.cs.umass.edu)? What is the Internet address of the computer that sent the HTTP GET message (i.e., your computer)?

Answer: Since the website is a https secured website, we use the dns filter, rather than the http filter on the wireshark protocols list. The DNS query and response details are summarised in Table 4 and can be seen in Figure 2.

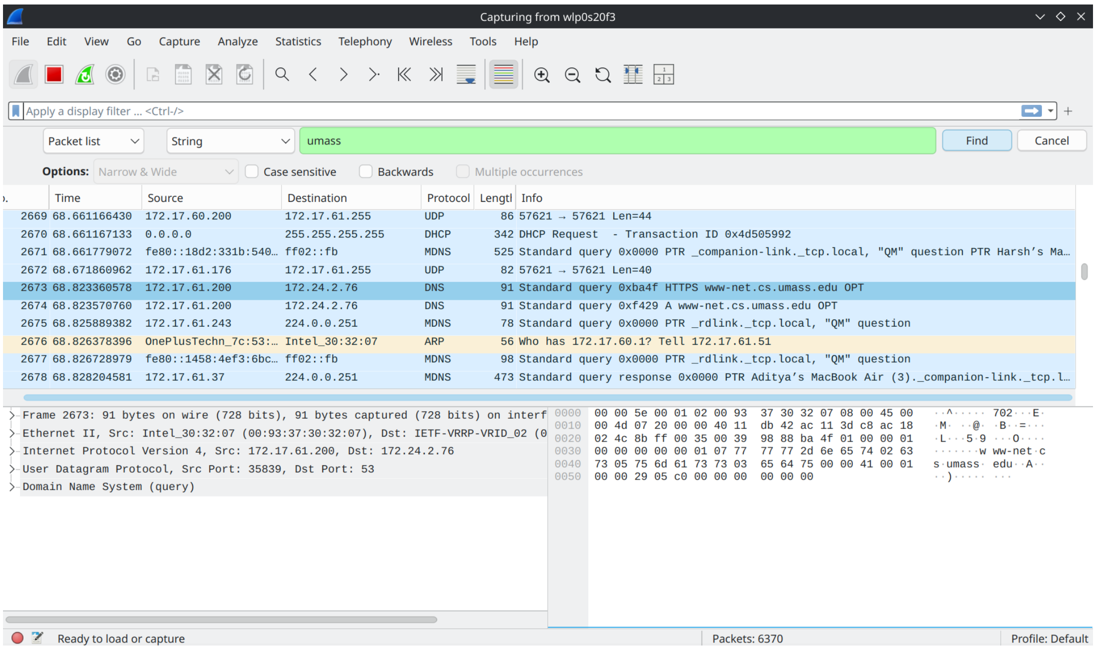

Figure 2: Screen Capture from Wireshark for the Website DNS Request

Field	**Value**
Query Name	`gaia.cs.umass.edu`
Query Packet No.	2673
Response Packet No.	2720
Query Host	172.17.61.200
DNS Server	172.24.2.76
Resolved Address	`128.119.245.12`
Response Time	1.989 ms

Table 4: DNS Query and Response Details

2.4 HTTP Details

Question: Expand the information on the HTTP message in the Wireshark "Details of selected packet" window (see Figure 2 of the Tutorial sheet) so you can see the fields in the HTTP GET request message. What type of Web browser issued the HTTP request? The answer is shown at the right end of the information following the "User-Agent:" field in the expanded HTTP message display. [This field value in the HTTP message is how a web server learns what type of browser you are using.]

Answer: You can see in Figure 3 that the user agent field is populated with the browser I used, which is Mozilla Firefox. My operating system and system type is also sent, as summarised below.

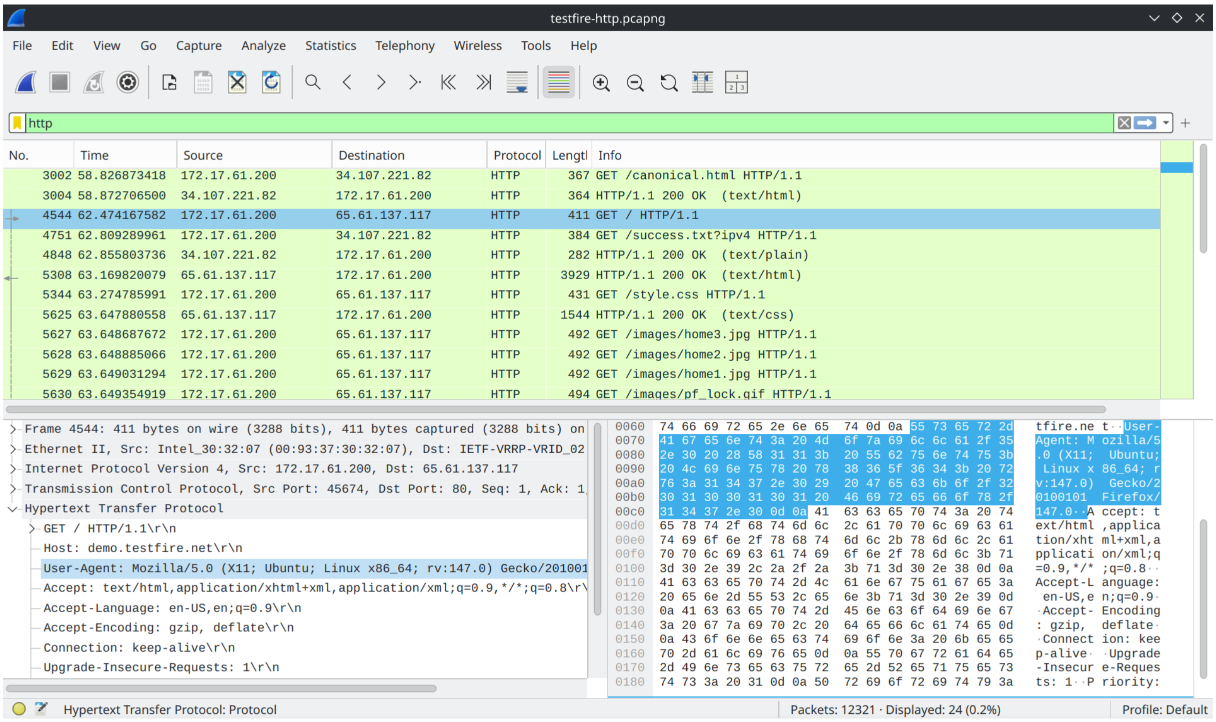

Figure 3: HTTP Browser Request Details

Field	Value
Browser	Mozilla Firefox
Operating System	Ubuntu
Architecture	Linux_x86_64

Table 5: HTTP User-Agent Details

2.5 TCP Stats

Question: Expand the information on the Transmission Control Protocol (TCP is a transport layer protocol, reference: Section 8.5 in Garcia) for this packet in the Wireshark "Details of selected packet" window so you can see the fields in the TCP segment carrying the HTTP message. What is the destination port number (the number following "Dest Port:" for the TCP segment containing the HTTP request) to which this HTTP request is being sent?

Answer: The port numbers for this HTTP request, as can be seen in Figure 4, are shown in Table 6.

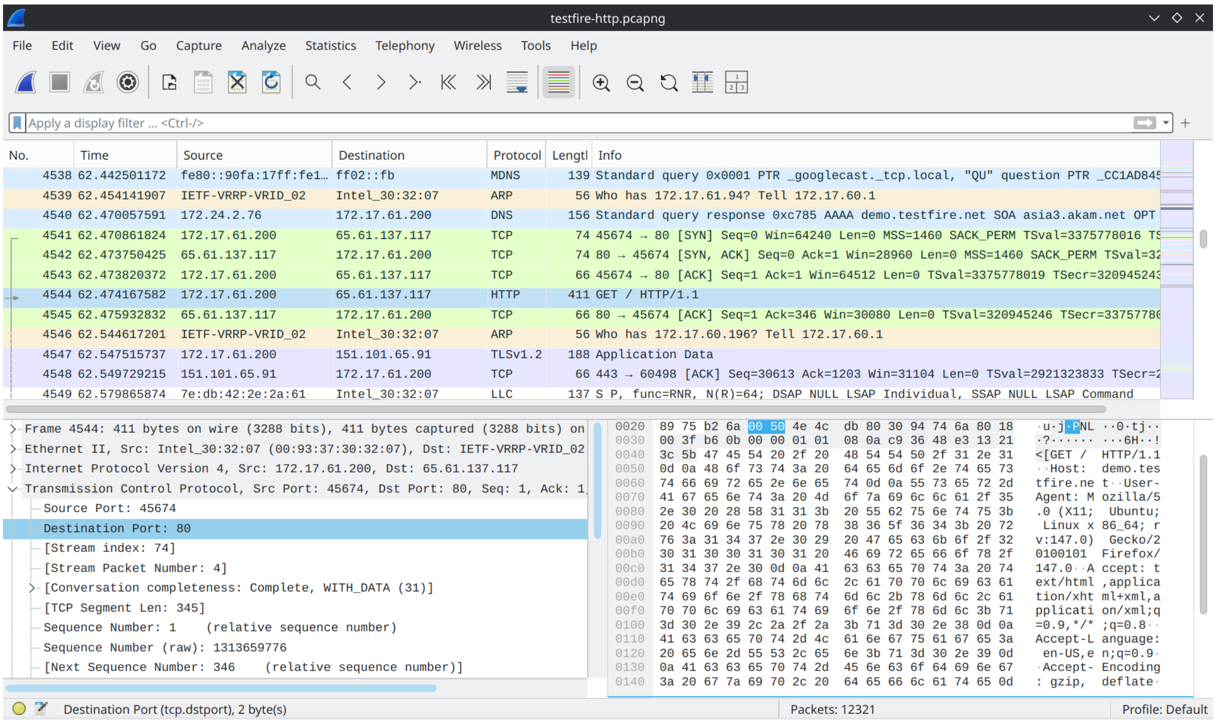

Figure 4: TCP Packet Showing Destination Port

Field	Value
Source Port	45675
Destination Port	80

Table 6: TCP Port Numbers for the HTTP Request

Note that, in the representation of the packet, you see the port number represented as `00 50`, but that's because this value is in hex. In decimal the packet number becomes 80.

2.6 Printing the HTTP Requests

Question: Print the two HTTP messages (GET and OK) referred to in question 2 above. To do so, select Print from the Wireshark File command menu, and select the "Selected Packet Only" and "Print as displayed" radial buttons, and then click OK.

Answer: The details of the HTTP packets have been printed, and the pdf is appended to the next page.

```
No.     Time           Source                Destination           Protocol Length Info
   4751 62.809289961   172.17.61.200         34.107.221.82         HTTP     384    GET /success.txt?ipv4 HTTP/1.1
Frame 4751: 384 bytes on wire (3072 bits), 384 bytes captured (3072 bits) on interface wlp0s20f3, id 0
Ethernet II, Src: Intel_30:32:07 (00:93:37:30:32:07), Dst: IETF-VRRP-VRID_02 (00:00:5e:00:01:02)
Internet Protocol Version 4, Src: 172.17.61.200, Dst: 34.107.221.82
Transmission Control Protocol, Src Port: 36004, Dst Port: 80, Seq: 1, Ack: 1, Len: 318
    Source Port: 36004
    Destination Port: 80
    [Stream index: 77]
    [Stream Packet Number: 4]
    [Conversation completeness: Complete, WITH_DATA (31)]
    [TCP Segment Len: 318]
    Sequence Number: 1    (relative sequence number)
    Sequence Number (raw): 3279544848
    [Next Sequence Number: 319    (relative sequence number)]
    Acknowledgment Number: 1    (relative ack number)
    Acknowledgment number (raw): 1653599650
    1000 .... = Header Length: 32 bytes (8)
    Flags: 0x018 (PSH, ACK)
    Window: 63
    [Calculated window size: 64512]
    [Window size scaling factor: 1024]
    Checksum: 0xeafb [unverified]
    [Checksum Status: Unverified]
    Urgent Pointer: 0
    Options: (12 bytes), No-Operation (NOP), No-Operation (NOP), Timestamps
    [Timestamps]
    [SEQ/ACK analysis]
    TCP payload (318 bytes)
Hypertext Transfer Protocol
No.     Time           Source                Destination           Protocol Length Info
   4848 62.855803736   34.107.221.82         172.17.61.200         HTTP     282    HTTP/1.1 200 OK  (text/plain)
Frame 4848: 282 bytes on wire (2256 bits), 282 bytes captured (2256 bits) on interface wlp0s20f3, id 0
Ethernet II, Src: JuniperNetwo_b6:d7:f0 (7c:e2:ca:b6:d7:f0), Dst: Intel_30:32:07 (00:93:37:30:32:07)
Internet Protocol Version 4, Src: 34.107.221.82, Dst: 172.17.61.200
Transmission Control Protocol, Src Port: 80, Dst Port: 36004, Seq: 1, Ack: 319, Len: 216
    Source Port: 80
    Destination Port: 36004
    [Stream index: 77]
    [Stream Packet Number: 6]
    [Conversation completeness: Complete, WITH_DATA (31)]
    [TCP Segment Len: 216]
    Sequence Number: 1    (relative sequence number)
    Sequence Number (raw): 1653599650
    [Next Sequence Number: 217    (relative sequence number)]
    Acknowledgment Number: 319    (relative ack number)
    Acknowledgment number (raw): 3279545166
    1000 .... = Header Length: 32 bytes (8)
    Flags: 0x018 (PSH, ACK)
    Window: 235
    [Calculated window size: 30080]
    [Window size scaling factor: 128]
    Checksum: 0x281a [unverified]
    [Checksum Status: Unverified]
    Urgent Pointer: 0
    Options: (12 bytes), No-Operation (NOP), No-Operation (NOP), Timestamps
    [Timestamps]
    [SEQ/ACK analysis]
    TCP payload (216 bytes)
Hypertext Transfer Protocol
Line-based text data: text/plain (1 lines)
```