

Q.1 — Configuration of the device group can best be understood in terms of what is happening “under the hood” (*i.e.*, beneath the surface, or behind the interface which the GUI provides). Describe the activities “under the hood” in terms which you have learnt in your study of the OSI 7-layer model.

Answer

So, firstly, it is critical to notice that this completely depends on what Ostinato does “under the hood”. Specifically, I would like to point out that Ostinato itself is an application running in a Linux virtual machine which is virtually connected to a docker container on the same machine. This of course introduces another level of complexity which I do not believe is feasible to consider here.

Also, I believe it is also not feasible to describe what Ostinato precisely does internally as it requires an in depth knowledge of the Ostinato codebase.

Having said this, I will restrict myself to a very shallow explanation.

The two main OSI layers which concern the creation of device groups are the Data Link (2nd) Layer and the Network (3rd) Layer.

When we create a device group we are forced to give that device group a base MAC (Media Access Control) address. We have to pick also the number of devices which the device group will have. In the case that there are more than one devices, Ostinato will give the rest of the devices MAC addresses based on an address offset which we specify. The MAC addresses are then used by Ostinato to distinguish between these virtual devices at layer 2.

Note: The uniqueness of the MAC addresses used is responsibility of the individual using Ostinato, although Ostinato seems to provide random MAC address which are unlikely to be already in use.

After we provide MAC addresses to the devices in the device group we also have the option of choosing an Internet Protocol (IP) version and provide a base IP address and an address offset to give each device a different IP address. This essentially sets up the virtual devices for layer 3 functionality.

I think the above is a sufficient answer as delving into the actual details of how these things are implemented is not trivial and would require an incredibly large amount of work.

Q.2.1 — Count the number of packets that you have captured and compare this with the setting in the stream configured on the packet generator.

Answer

Table 1: Wireshark ICMP capture for Question 2.1.

No.	Time	Source	Destination	Protocol	Length	Info
486	4617.638757	192.168.0.10	192.168.0.1	ICMP	60	Echo (ping) request id=0x04d2, seq=0/0, ttl=127 (reply in 487)
continued on next page						

Table 1 – continued from previous page

No.	Time	Source	Destination	Protocol	Length	Info
487	4617.638893	192.168.0.1	192.168.0.10	ICMP	60	Echo (ping) reply id=0x04d2, seq=0/0, ttl=64 (request in 486)
488	4618.638764	192.168.0.10	192.168.0.1	ICMP	60	Echo (ping) request id=0x04d2, seq=0/0, ttl=127 (reply in 489)
489	4618.638848	192.168.0.1	192.168.0.10	ICMP	60	Echo (ping) reply id=0x04d2, seq=0/0, ttl=64 (request in 488)
490	4619.638761	192.168.0.10	192.168.0.1	ICMP	60	Echo (ping) request id=0x04d2, seq=0/0, ttl=127 (reply in 491)
491	4619.638880	192.168.0.1	192.168.0.10	ICMP	60	Echo (ping) reply id=0x04d2, seq=0/0, ttl=64 (request in 490)
492	4620.638772	192.168.0.10	192.168.0.1	ICMP	60	Echo (ping) request id=0x04d2, seq=0/0, ttl=127 (reply in 493)
493	4620.638890	192.168.0.1	192.168.0.10	ICMP	60	Echo (ping) reply id=0x04d2, seq=0/0, ttl=64 (request in 492)
494	4621.638731	192.168.0.10	192.168.0.1	ICMP	60	Echo (ping) request id=0x04d2, seq=0/0, ttl=127 (reply in 495)
495	4621.638861	192.168.0.1	192.168.0.10	ICMP	60	Echo (ping) reply id=0x04d2, seq=0/0, ttl=64 (request in 494)
496	4622.638714	192.168.0.10	192.168.0.1	ICMP	60	Echo (ping) request id=0x04d2, seq=0/0, ttl=127 (reply in 497)
497	4622.638914	192.168.0.1	192.168.0.10	ICMP	60	Echo (ping) reply id=0x04d2, seq=0/0, ttl=64 (request in 496)
500	4623.638729	192.168.0.10	192.168.0.1	ICMP	60	Echo (ping) request id=0x04d2, seq=0/0, ttl=127 (reply in 501)
501	4623.638837	192.168.0.1	192.168.0.10	ICMP	60	Echo (ping) reply id=0x04d2, seq=0/0, ttl=64 (request in 500)
502	4624.638727	192.168.0.10	192.168.0.1	ICMP	60	Echo (ping) request id=0x04d2, seq=0/0, ttl=127 (reply in 503)
503	4624.638831	192.168.0.1	192.168.0.10	ICMP	60	Echo (ping) reply id=0x04d2, seq=0/0, ttl=64 (request in 502)

continued on next page

Table 1 – continued from previous page

No.	Time	Source	Destination	Protocol	Length	Info
504	4625.638735	192.168.0.10	192.168.0.1	ICMP	60	Echo (ping) request id=0x04d2, seq=0/0, ttl=127 (reply in 505)
505	4625.638858	192.168.0.1	192.168.0.10	ICMP	60	Echo (ping) reply id=0x04d2, seq=0/0, ttl=64 (request in 504)
506	4626.638725	192.168.0.10	192.168.0.1	ICMP	60	Echo (ping) request id=0x04d2, seq=0/0, ttl=127 (reply in 507)
507	4626.638833	192.168.0.1	192.168.0.10	ICMP	60	Echo (ping) reply id=0x04d2, seq=0/0, ttl=64 (request in 506)

There are 20 ICMP packets. 10 requests and 10 replies.

The screenshot shows the Ostinato configuration window for a stream. The 'Send' tab is selected, and the 'Packets' radio button is chosen. The 'Number of Packets' is set to 10. The 'Mode' is set to 'Fixed'. The 'Rate' is set to 1.0000 Packets/Sec. The 'After this stream' section has 'Goto Next Stream' selected. The 'Gaps (in seconds)' section shows ISG, IBG, and IPG all set to 0.0.

Figure 1: The stream settings in Ostinato for Question 2.1.

As can be seen in Figure 1 the number of packets, specifically requests, was precisely set to 10.

Q.2.2 — Write down the numbers (leftmost column) of the packets that have been generated by:

- the packet generator
- the DUT

Answer

The required values can be read off Table 1.

486, 488, 490, 492, 494, 496, 500, 502, 504 and 506 are the numbers associated with the request packets generated by Ostinato.

487, 489, 491, 493, 495, 497, 501, 503, 505 and 507 are the numbers associated with the reply packets sent by the DUT.

Q.2.3 — Use your answer to 2.2 to find the average inter-packet delay for the packets generated by the packet generator.

Answer

$$\begin{aligned} & (4618.638764 - 4617.638757) + (4619.638761 - 4618.638764) + (4620.638772 - 4619.638761) \\ & + (4621.638731 - 4620.638772) + (4622.638714 - 4621.638731) + (4623.638729 - 4622.638714) \\ & + (4624.638727 - 4623.638729) + (4625.638735 - 4624.638727) + (4626.638725 - 4625.638735) \\ & \quad \quad \quad 10 - 1 \\ & = 0.999996444444535 \text{ seconds} \\ & \approx 1 \text{ seconds} \end{aligned}$$

The above computed value is the average inter-packet delay (in seconds). The approximate value, 1, is precisely what was set in the stream's settings, see Figure 1.

Listing 1: Python expression for calculating the inter-packet delay for Question 2.3.

```
1 ((4618.638764 - 4617.638757)
2 + (4619.638761 - 4618.638764)
3 + (4620.638772 - 4619.638761)
4 + (4621.638731 - 4620.638772)
5 + (4622.638714 - 4621.638731)
6 + (4623.638729 - 4622.638714)
7 + (4624.638727 - 4623.638729)
8 + (4625.638735 - 4624.638727)
9 + (4626.638725 - 4625.638735)) / 9
```

Q.3 — Repeat the exercises of question 2 with the new packet rate (2 pkt/s).

Answer

The screenshot shows the 'Edit Stream [Echo Request]' dialog box. The 'Send' tab is selected. Under 'Send', 'Packets' is chosen. Under 'Mode', 'Fixed' is chosen. Under 'Rate', 'Packets/Sec' is chosen with a value of 2.0000. In the 'Numbers' section, 'Number of Packets' is 10 and 'Packets per Burst' is 10. In the 'After this stream' section, 'Goto Next Stream' is selected. The 'Gaps (in seconds)' section shows ISG as 0.0, IBG as 0.0, and IPG as 0.500000000. Buttons for 'Prev', 'Next', 'OK', and 'Cancel' are at the bottom.

Figure 2: The stream settings in Ostinato for Question 3

Table 2: Wireshark ICMP capture for Question 3.

No.	Time	Source	Destination	Protocol	Length	Info
16	140.926179	192.168.0.10	192.168.0.1	ICMP	60	Echo (ping) request id=0x04d2, seq=0/0, ttl=127 (reply in 17)
17	140.926306	192.168.0.1	192.168.0.10	ICMP	60	Echo (ping) reply id=0x04d2, seq=0/0, ttl=64 (request in 16)
18	141.426171	192.168.0.10	192.168.0.1	ICMP	60	Echo (ping) request id=0x04d2, seq=0/0, ttl=127 (reply in 19)
19	141.426320	192.168.0.1	192.168.0.10	ICMP	60	Echo (ping) reply id=0x04d2, seq=0/0, ttl=64 (request in 18)
20	141.926165	192.168.0.10	192.168.0.1	ICMP	60	Echo (ping) request id=0x04d2, seq=0/0, ttl=127 (reply in 21)
21	141.926287	192.168.0.1	192.168.0.10	ICMP	60	Echo (ping) reply id=0x04d2, seq=0/0, ttl=64 (request in 20)
continued on next page						

Table 2 – continued from previous page

No.	Time	Source	Destination	Protocol	Length	Info
22	142.426145	192.168.0.10	192.168.0.1	ICMP	60	Echo (ping) request id=0x04d2, seq=0/0, ttl=127 (reply in 23)
23	142.426254	192.168.0.1	192.168.0.10	ICMP	60	Echo (ping) reply id=0x04d2, seq=0/0, ttl=64 (request in 22)
24	142.926106	192.168.0.10	192.168.0.1	ICMP	60	Echo (ping) request id=0x04d2, seq=0/0, ttl=127 (reply in 25)
25	142.926275	192.168.0.1	192.168.0.10	ICMP	60	Echo (ping) reply id=0x04d2, seq=0/0, ttl=64 (request in 24)
26	143.426150	192.168.0.10	192.168.0.1	ICMP	60	Echo (ping) request id=0x04d2, seq=0/0, ttl=127 (reply in 27)
27	143.426249	192.168.0.1	192.168.0.10	ICMP	60	Echo (ping) reply id=0x04d2, seq=0/0, ttl=64 (request in 26)
28	143.926104	192.168.0.10	192.168.0.1	ICMP	60	Echo (ping) request id=0x04d2, seq=0/0, ttl=127 (reply in 29)
29	143.926259	192.168.0.1	192.168.0.10	ICMP	60	Echo (ping) reply id=0x04d2, seq=0/0, ttl=64 (request in 28)
30	144.426138	192.168.0.10	192.168.0.1	ICMP	60	Echo (ping) request id=0x04d2, seq=0/0, ttl=127 (reply in 31)
31	144.426250	192.168.0.1	192.168.0.10	ICMP	60	Echo (ping) reply id=0x04d2, seq=0/0, ttl=64 (request in 30)
32	144.926075	192.168.0.10	192.168.0.1	ICMP	60	Echo (ping) request id=0x04d2, seq=0/0, ttl=127 (reply in 33)
33	144.926259	192.168.0.1	192.168.0.10	ICMP	60	Echo (ping) reply id=0x04d2, seq=0/0, ttl=64 (request in 32)
34	145.426128	192.168.0.10	192.168.0.1	ICMP	60	Echo (ping) request id=0x04d2, seq=0/0, ttl=127 (reply in 35)
35	145.426236	192.168.0.1	192.168.0.10	ICMP	60	Echo (ping) reply id=0x04d2, seq=0/0, ttl=64 (request in 34)

Again the number of packets in total is 20. 10 requests and 10 replies as specified in the stream's settings, see Figure 2.

And from Table 2 we can get the respective No. of the request and reply packets

The request packets are precisely numbers: 16, 18, 20, 22, 24, 26, 28, 30, 32, 34.

And the reply packets are precisely numbers: 17, 19, 21, 23, 25, 27, 29, 31, 33, 35.

Finally, the below computed value is the average inter-packet delay (in seconds).

$$\frac{(141.426171 - 140.926179) + (141.926165 - 141.426171) + (142.426145 - 141.926165) + (142.926106 - 142.426145) + (143.426150 - 142.926106) + (143.926104 - 143.426150) + (144.426138 - 143.926104) + (144.926075 - 144.426138) + (145.426128 - 144.926075)}{10 - 1}$$

= 0.499994333333335 seconds

≈ 0.5 seconds

The approximate value, 0.5, is precisely what should be expected since if two request are being sent per second, see Figure 2, that would equate to a packer every half a second which is precisely 0.5 seconds.

Listing 2: Python expression for calculating the inter-packet delay for Question 3.

```
1 ((141.426171 - 140.926179)
2 + (141.926165 - 141.426171)
3 + (142.426145 - 141.926165)
4 + (142.926106 - 142.426145)
5 + (143.426150 - 142.926106)
6 + (143.926104 - 143.426150)
7 + (144.426138 - 143.926104)
8 + (144.926075 - 144.426138)
9 + (145.426128 - 144.926075)) / 9
```

Q.4.1 — Expand the frame section and inspect it to find out the number of bytes captured by Wireshark from the link (“bytes on wire”).

Answer

1	0.000000	192.168.0.10	192.168.0.1	ICMP	60 Echo (ping) request	id=0x04d2, seq=0/0, ttl=127 (reply in 2)
2	0.000299	192.168.0.1	192.168.0.10	ICMP	60 Echo (ping) reply	id=0x04d2, seq=0/0, ttl=64 (request in 1)

Figure 3: The packets under inspection for Question 4.1.

The info related to the request from the generator to the DUT is provided below. Specifically, the information related to the Ethernet II frame.

```

▼ Frame 1: 60 bytes on wire (480 bits), 60 bytes captured (480 bits) on interface -, id 0
  Section number: 1
  ▶ Interface id: 0 (-)
    Encapsulation type: Ethernet (1)
    Arrival Time: Dec  3, 2023 11:51:36.046898000 CET
    UTC Arrival Time: Dec  3, 2023 10:51:36.046898000 UTC
    Epoch Arrival Time: 1701600696.046898000
    [Time shift for this packet: 0.000000000 seconds]
    [Time delta from previous captured frame: 0.000000000 seconds]
    [Time delta from previous displayed frame: 0.000000000 seconds]
    [Time since reference or first frame: 0.000000000 seconds]
    Frame Number: 1
    Frame Length: 60 bytes (480 bits)
    Capture Length: 60 bytes (480 bits)
    [Frame is marked: False]
    [Frame is ignored: False]
    [Protocols in frame: eth:ethertype:ip:icmp:data]
    [Coloring Rule Name: ICMP]
    [Coloring Rule String: icmp || icmpv6]
  ▶ Ethernet II, Src: 90:00:01:a9:41:f1 (90:00:01:a9:41:f1), Dst: be:98:b5:5f:fb:a8 (be:98:b5:5f:fb:a8)
  ▶ Internet Protocol Version 4, Src: 192.168.0.10, Dst: 192.168.0.1
  ▶ Internet Control Message Protocol

```

Figure 4: The frame information of the request packet under inception for Question 4.1.

The number of bytes capture by Wireshark is exactly 60 bytes.

Q.4.2 — Search for the minimum length of an Ethernet packet, and state it.

Answer

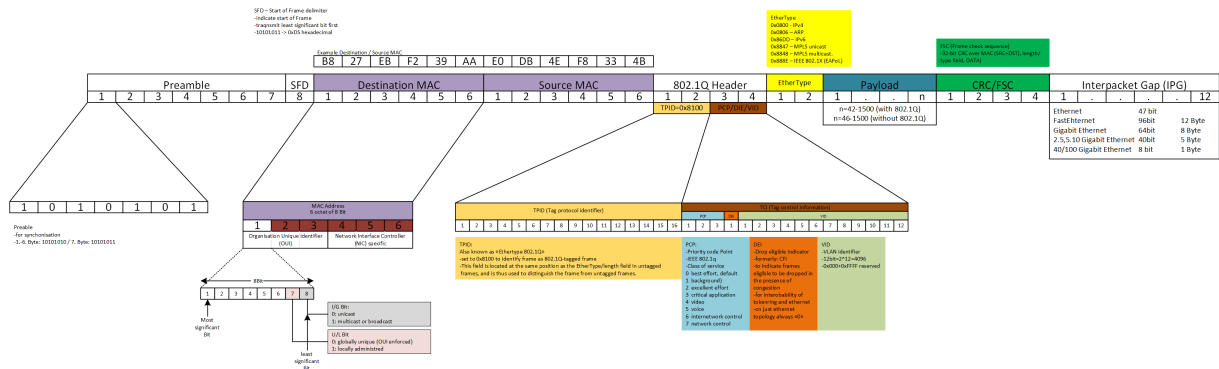


Figure 5: A diagram of the components of an Ethernet II frame.

Reference: https://upload.wikimedia.org/wikipedia/commons/7/72/Ethernet_Frame.png

Note: There is a mistake in the graphic. “CRC/FSC” should be “CRC/FCS” and “FSC (Frame check sequence)” should be “FCS (Frame check sequence)”.

The minimum and maximum lengths can be derived from Figure 5. Importantly, the preamble, SFD and inter-pack gap are almost never exposed beyond layer 1. Therefore, we shall not factor these into our calculation.

Hence, the Ethernet II frame at layer 2 consists of the following segments:

- Destination MAC (6 octets);
- Source MAC (6 octets);
- 802.1Q Header (optional & 4 octets);
- EtherType (2 octets);
- Payload (42 — 1500 octets with 802.1Q & 46 — 1500 octets without 802.1Q); and
- CRC/FCS (4 octets).

Following this, the minimum length can be calculated follows:

$$\text{Frame}_{\min} = 6 + 6 + 2 + 46 + 4 = 64 \text{ octets}$$

Note: In the case when there is no 802.1Q header, the total header and payload lengths add up to 46. This is because the header length is 0 and the payload has a minimum length of 46. In the other case *i.e.* when there is a 802.1Q header, the total header and payload lengths also add up to 46. This is because the header length is 4 and the payload length has a new minimum of 42, since 4 have already been used by 802.1Q header. Hence, both cases the sum of their lengths is identical.

Similarly, the maximum length is given by the following calculation:

$$\text{Frame}_{\max} = 6 + 6 + 2 + 1500 + 4 = 1518 \text{ octets}$$

Q.4.3 — How does the number of bytes captured (which you found in (4.1)) compare with the minimum length of an Ethernet packet (which you looked up in (4.2))? Can you explain the difference?

Answer

```

▶ Frame 1: 60 bytes on wire (480 bits), 60 bytes captured (480 bits) on interface -, id 0
▼ Ethernet II, Src: 90:00:01:a9:41:f1 (90:00:01:a9:41:f1), Dst: be:98:b5:5f:fb:a8 (be:98:b5:5f:fb:a8)
  ▶ Destination: be:98:b5:5f:fb:a8 (be:98:b5:5f:fb:a8)
  ▶ Source: 90:00:01:a9:41:f1 (90:00:01:a9:41:f1)
  ▶ Type: IPv4 (0x0800)
▶ Internet Protocol Version 4, Src: 192.168.0.10, Dst: 192.168.0.1
▶ Internet Control Message Protocol

```

Figure 6: The Ethernet II header of the request packet under inspection for Question 4.1.

The number of “bytes on wire” according to Wireshark, see Figure 6, is precisely 60 bytes. However, the minimum number of bytes is at least 64. This of course, is a 4 byte discrepancy.

To account for this discrepancy, notice that the Ethernet II header does not contain an 802.1Q header. Hence, the payload is allowed a minimum of 46 bytes. Additionally, all these 46 bytes are used up by the ICMP request (including all IP overhead).

Hence, the Ethernet header and ICMP request sum up to 60 bytes. This leaves only one option as to what the remaining 4 bytes can they constitute the Cyclic Redundancy Check (CRC) or Frame Check Sequence (FCS). In fact, according to Figure 5, CRC field is exactly 4 bytes.

Furthermore, this conclusion is further supported by the below referenced Wireshark forum discussion. One of the users exclaims that “bytes on wire” is often more like “bytes on wire without CRC”. This is the case because most network drivers do not provide the CRC to user space applications, instead invalid packets are just immediately dropped.

Reference: <https://osqa-ask.wireshark.org/questions/1344/does-frame-length-include-also-crc-bytes>

Q.4.4 — Expand the Ethernet II section and write down the source MAC address and the destination MAC address.

Answer

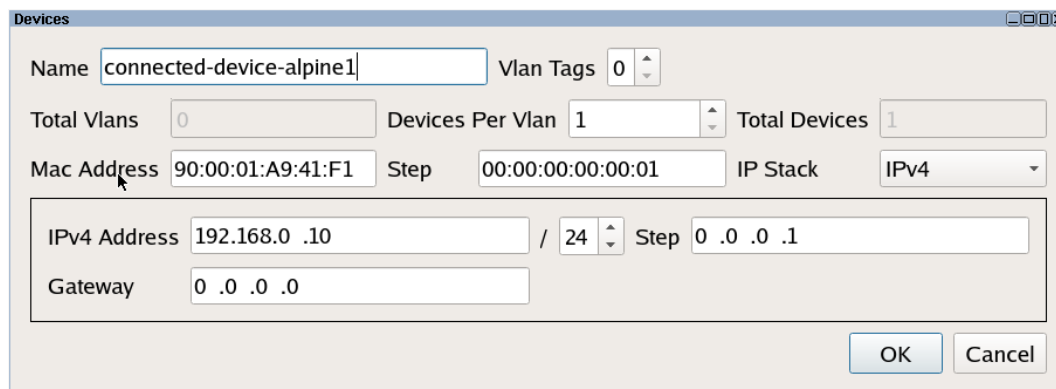
Source MAC = 90:00:01:a9:41:f1

Destination MAC = be:98:b5:5f:fb:a8

The above where taken from Figure 6.

Q.4.5 — Look up the source MAC address in the device group configured on the packet generator at the port group.

Answer



The screenshot shows the 'Devices' configuration window in Ostinator. The 'Name' field is 'connected-device-alpine1'. 'Vlan Tags' is 0. 'Total Vlans' is 0, 'Devices Per Vlan' is 1, and 'Total Devices' is 1. The 'Mac Address' field is '90:00:01:A9:41:F1'. The 'Step' field is '00:00:00:00:00:01'. The 'IP Stack' is set to 'IPv4'. Below this, the 'IPv4 Address' is '192.168.0 .10' with a subnet mask of '24' and a 'Step' of '0 .0 .0 .1'. The 'Gateway' is '0 .0 .0 .0'. 'OK' and 'Cancel' buttons are at the bottom right.

Figure 7: The device group configuration in Ostinator for Question 4.5.

Note: Since the device group contains as single device the base MAC address *i.e.* 90:00:01:a9:41:f1 is used for that device.

As can be seen in Figure 7, the device is given the MAC address 90:00:01:a9:41:f1.

Q.4.6 — Compare what was found in 4.4 with what was found in 4.5.

Answer

Note: Assuming that in the original question, 5.4 and 5.5 were meant to be 4.4 and 4.5 respectively.

As expected, the MAC address of the virtual device is identical to the source MAC Address in the packet. This is because the packet is a request packet *i.e.* it was created by Ostinato.

It is also critical to notice that the request is very much dependent on ARP (Address Resolution Protocol) to establish who has which MAC addresses.

Q.4.7 — Inspect the Ethernet II section and find the field in that section which the receiver (the DUT) uses to identify the layer 3 entity which is to receive the Ethernet frame's payload.

Answer

In Question 4.2 the EtherType field is referenced. The EtherType field specifies what type of payload the Ethernet frame contains. The EtherType in the frame of interest is set to 0x0800, see Figure 6. 0x0800 identifies the payload inside the frame as an IPv4 packet. This gives the DUT the required information to properly decode the contents of the payload.

Q.4.8 — Inspect the section below the Ethernet II section and:

- write down the source address and the destination address that you see in this underlying section;
- compare the source address and the destination address with what you have set up in the stream named “ICMP Echo Request Stream”.
- Look up the field named “Total length” in this section and account for the difference between this number and the number you have found in 4.1.

Answer

```
▶ Frame 1: 60 bytes on wire (480 bits), 60 bytes captured (480 bits) on interface -, id 0
▶ Ethernet II, Src: 90:00:01:a9:41:f1 (90:00:01:a9:41:f1), Dst: be:98:b5:5f:fb:a8 (be:98:b5:5f:fb:a8)
▼ Internet Protocol Version 4, Src: 192.168.0.10, Dst: 192.168.0.1
    0100 .... = Version: 4
    .... 0101 = Header Length: 20 bytes (5)
    ▶ Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
        Total Length: 46
        Identification: 0x04d2 (1234)
    ▶ 000. .... = Flags: 0x0
        ...0 0000 0000 0000 = Fragment Offset: 0
        Time to Live: 127
        Protocol: ICMP (1)
        Header Checksum: 0xb5a1 [validation disabled]
        [Header checksum status: Unverified]
        Source Address: 192.168.0.10
        Destination Address: 192.168.0.1
▶ Internet Control Message Protocol
```

Figure 8: The IPv4 header of the request packet under inspection for Question 4.1.

Source IP Address = 192.168.0.10

Destination IP Address = 192.168.0.1

See Figure 8 to verify the above addresses.

Protocol Selection	Protocol Data	Variable Fields	Stream Control	Packet View																				
Media Access Protocol																								
Ethernet II																								
Internet Protocol ver 4																								
<table border="1"> <thead> <tr> <th></th> <th></th> <th>Mode</th> <th>Count</th> <th>Mask</th> </tr> </thead> <tbody> <tr> <td>Source</td> <td>192.168.0 .10</td> <td>Fixed</td> <td>16</td> <td>255.255.255.0</td> </tr> <tr> <td>Destination</td> <td>192.168.0 .1</td> <td>Fixed</td> <td>16</td> <td>255.255.255.0</td> </tr> <tr> <td>Options</td> <td colspan="4"></td> </tr> </tbody> </table>							Mode	Count	Mask	Source	192.168.0 .10	Fixed	16	255.255.255.0	Destination	192.168.0 .1	Fixed	16	255.255.255.0	Options				
		Mode	Count	Mask																				
Source	192.168.0 .10	Fixed	16	255.255.255.0																				
Destination	192.168.0 .1	Fixed	16	255.255.255.0																				
Options																								
Internet Control Message Protocol																								
Payload Data																								

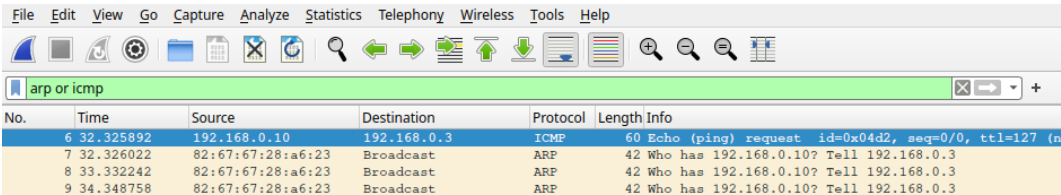
Figure 9: The stream's IPv4 configuration in Ostinato for Question 4.8.

Additionally, the Total Length is 46 bytes, again see Figure 8. This is precisely what was described in Question 4.3. Again, the Ethernet header length is 14 bytes, and $14 + 46 = 60$ bytes as expected.

Additionally, 20 of the 46 bytes are the IPv4 Header Length whilst the remaining 26 bytes are the actual ICMP Request.

Q.5 — For each packet, state source and destination IPv4 address. Compare these latter two addresses with the IPv4 address bound to alpine-1 eth0, and justify the outcome of your comparison.

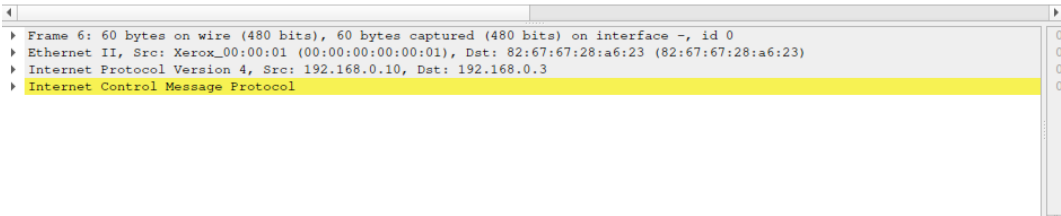
Answer



The image shows a Wireshark capture with the filter 'arp or icmp'. The packet list contains four entries:

No.	Time	Source	Destination	Protocol	Length	Info
6	32.325892	192.168.0.10	192.168.0.3	ICMP	60	Echo (ping) request id=0x04d2, seq=0/0, ttl=127 (n
7	32.326022	82:67:67:28:a6:23	Broadcast	ARP	42	Who has 192.168.0.10? Tell 192.168.0.3
8	33.332242	82:67:67:28:a6:23	Broadcast	ARP	42	Who has 192.168.0.10? Tell 192.168.0.3
9	34.348758	82:67:67:28:a6:23	Broadcast	ARP	42	Who has 192.168.0.10? Tell 192.168.0.3

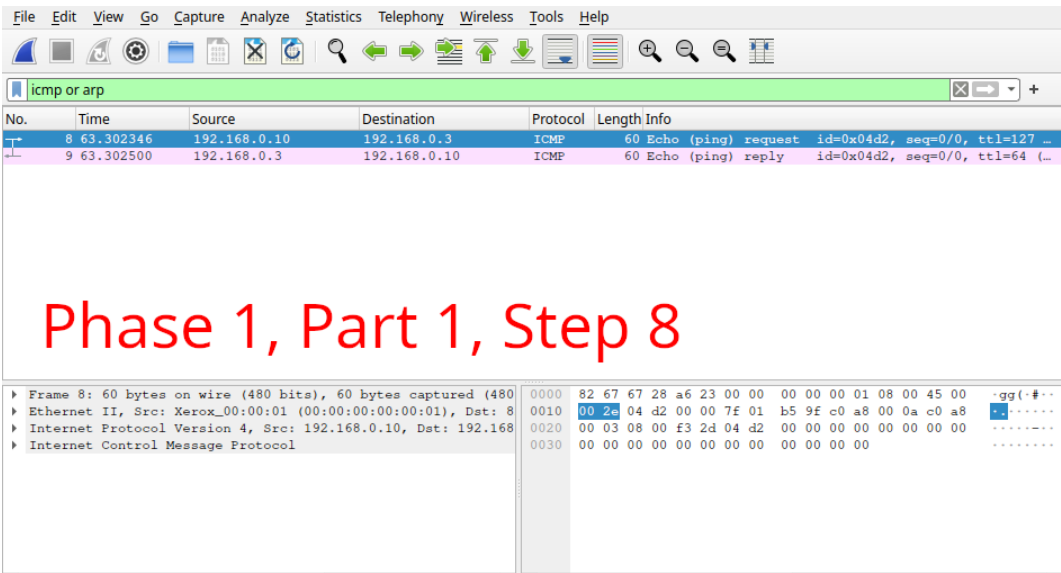
Phase 1, Part 1, Step 6



The image shows the packet details pane for packet 6. The expanded layers are:

- Frame 6: 60 bytes on wire (480 bits), 60 bytes captured (480 bits) on interface ~, id 0
- Ethernet II, Src: Xerox_00:00:01 (00:00:00:00:00:01), Dst: 82:67:67:28:a6:23 (82:67:67:28:a6:23)
- Internet Protocol Version 4, Src: 192.168.0.10, Dst: 192.168.0.3
- Internet Control Message Protocol

Figure 10: The Wireshark capture for phase 1, part 1, step 6.



The image shows a Wireshark capture with the filter 'icmp or arp'. The packet list contains two entries:

No.	Time	Source	Destination	Protocol	Length	Info
8	63.302346	192.168.0.10	192.168.0.3	ICMP	60	Echo (ping) request id=0x04d2, seq=0/0, ttl=127 ...
9	63.302500	192.168.0.3	192.168.0.10	ICMP	60	Echo (ping) reply id=0x04d2, seq=0/0, ttl=64 (...)

The packet details pane for packet 8 is expanded, showing the following layers:

- Frame 8: 60 bytes on wire (480 bits), 60 bytes captured (480 bits) on interface ~, id 0
- Ethernet II, Src: Xerox_00:00:01 (00:00:00:00:00:01), Dst: 82:67:67:28:a6:23 (82:67:67:28:a6:23)
- Internet Protocol Version 4, Src: 192.168.0.10, Dst: 192.168.0.3
- Internet Control Message Protocol

The packet bytes pane shows the raw data of the ICMP echo request:

```
0000  82 67 67 28 a6 23 00 00 00 00 00 01 08 00 45 00  -gg(-#...
0010  00 2e 04 d2 00 00 7f 01 b5 9f c0 a8 00 0a c0 a8  -...
0020  00 03 08 00 f3 2d 04 d2 00 00 00 00 00 00 00 00  -...
0030  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  -...
```

Figure 11: The Wireshark capture for phase 1, part 1, step 8.

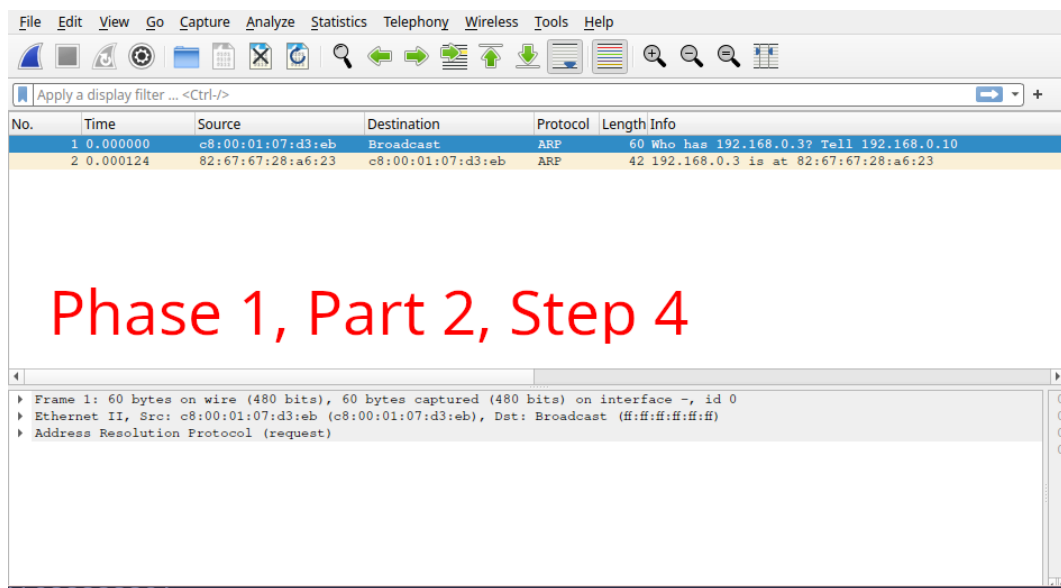


Figure 12: The Wireshark capture for phase 1, part 2, step 4.

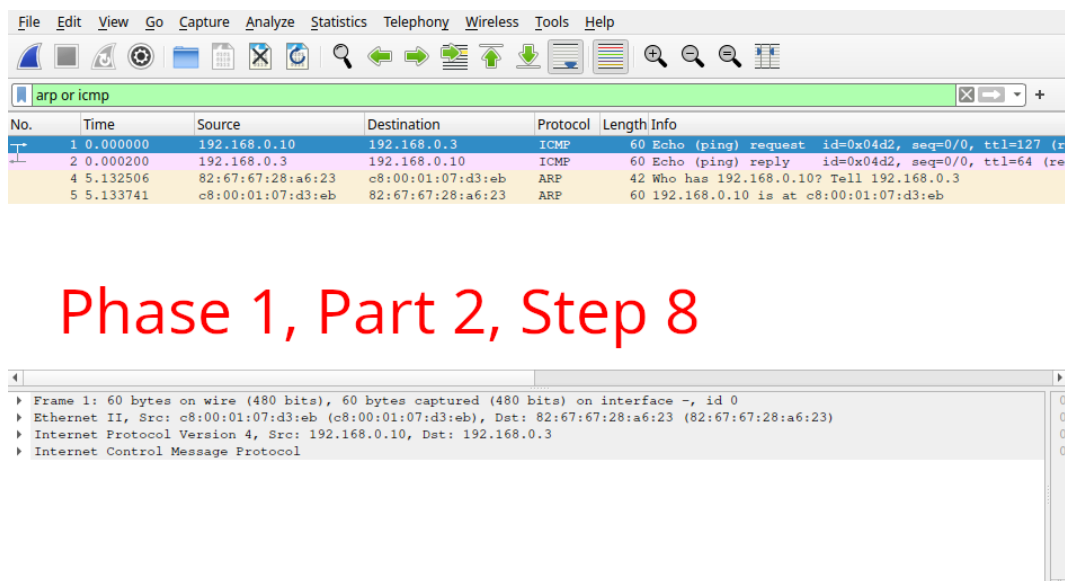


Figure 13: The Wireshark capture for phase 1, part 2, step 8.

The above four pictures are all the Wireshark captures as specified in the lab sheet.

All we have to do is notice that the addresses in use are From Figure 10 it can be deduced that 192.168.0.10 and 192.168.0.3 are the IP addresses in use. These are the IP address of the virtual Ostinato device and alpine-3 respectively.

Note: All the captures in Figures 10, 11, 12 & 13 were made over Hub1 ↔ alpine-1. This exposes the nature of hubs as networking devices. Hubs do not keep track of any form source and destination. Hubs take a broad stroke approach and replay any communication they receive into all of their connections.

Q.6 — Compare the packet(s) you capture on Switch1 ↔ alpine-2. State at least one difference between your output and that shown in Fig. 37, and explain it.

Answer

For this question all the steps described in Phase 1, Part 2 were repeated. However, this time Port 1 (in Ostinato) was used and connections Switch1 ↔ alpine-2 and Switch1 ↔ alpine-4 were monitored.

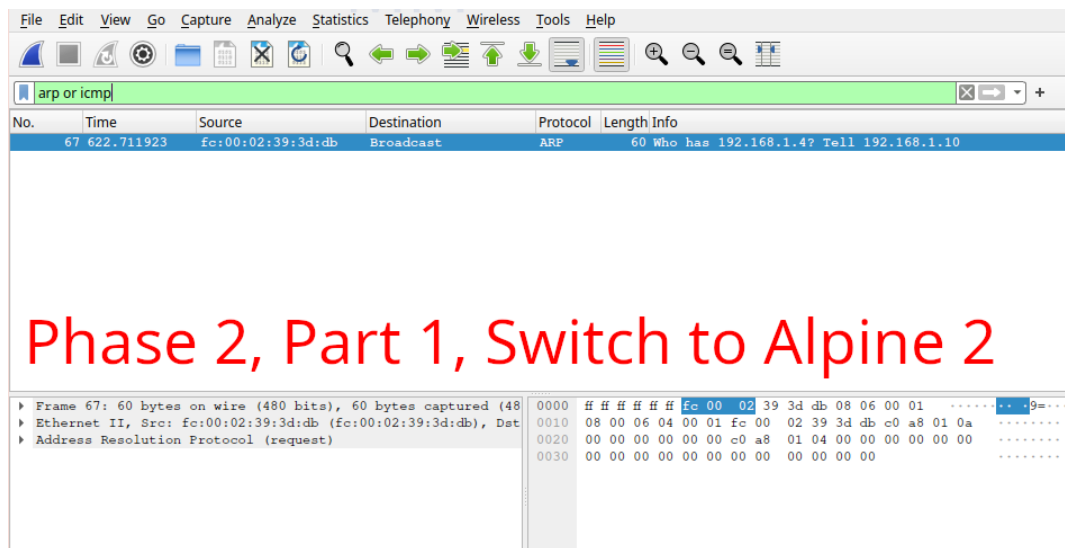
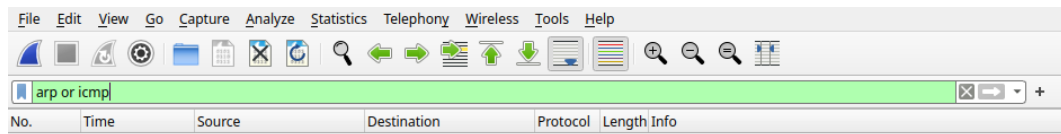


Figure 14: The Wireshark capture for phase 2, part 1 on connection Switch1 ↔ alpine-2.



Phase 2, Part 2, Switch to Alpine 2



Figure 15: The Wireshark capture for phase 2, part 2 on connection Switch1 ↔ alpine-2.

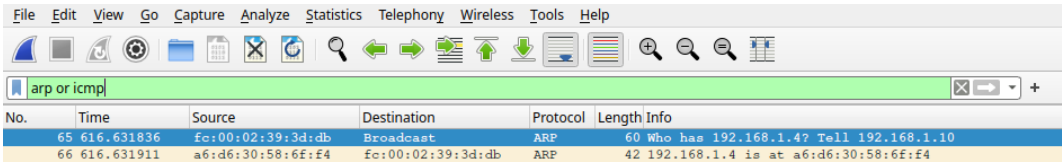
Firstly, it is important to note that only a single packet is captured on the Switch1 ↔ alpine-2 connection, see Figure 14 and 15.

Additionally, the only difference between the ARP packet in Figure 14 and the ARP packet in Fig. 37 of the lab sheet is the MAC address.

However, this could have also been made the same since the MAC address can be set by the user in *Ostinato*.

Q.7.1 — Present a screenshot that shows the packets you have captured on Switch1 ↔ alpine-4.

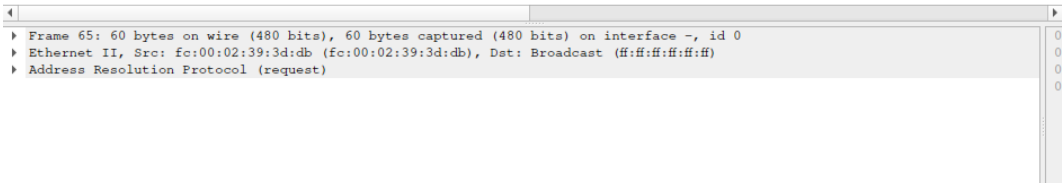
Answer



The screenshot shows a Wireshark capture with two packets. The first packet is an ARP request from the switch to the alpine host, and the second is the corresponding reply.

No.	Time	Source	Destination	Protocol	Length	Info
65	616.631836	fc:00:02:39:3d:db	Broadcast	ARP	60	Who has 192.168.1.4? Tell 192.168.1.10
66	616.631911	a6:d6:30:58:6f:f4	fc:00:02:39:3d:db	ARP	42	192.168.1.4 is at a6:d6:30:58:6f:f4

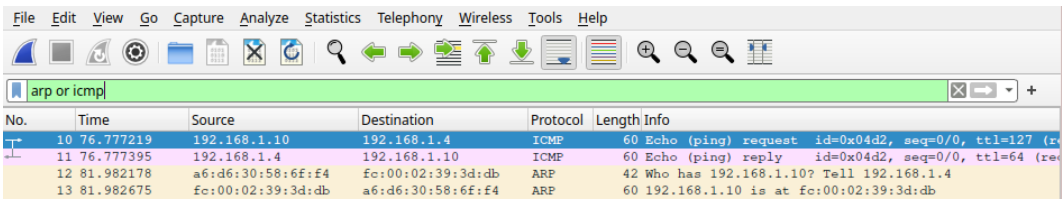
Phase 2, Part 1, Switch to Alpine 4



The screenshot shows the packet details for the first ARP request. It is an Ethernet II frame with source MAC fc:00:02:39:3d:db and destination MAC ff:ff:ff:ff:ff:ff (broadcast). The data is an ARP request.

Frame 65: 60 bytes on wire (480 bits), 60 bytes captured (480 bits) on interface -, id 0
Ethernet II, Src: fc:00:02:39:3d:db (fc:00:02:39:3d:db), Dst: Broadcast (ff:ff:ff:ff:ff:ff)
Address Resolution Protocol (request)

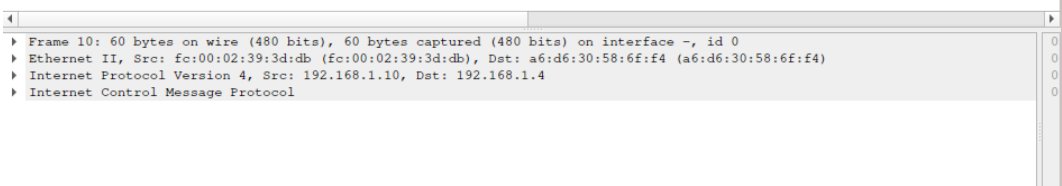
Figure 16: The Wireshark capture for phase 2, part 1 on connection Switch1 ↔ alpine-4.



The screenshot shows a Wireshark capture with four packets. The first two are ICMP echo request and reply between alpine-4 and the switch. The next two are ARP request and reply from the switch to alpine-4.

No.	Time	Source	Destination	Protocol	Length	Info
10	76.777219	192.168.1.10	192.168.1.4	ICMP	60	Echo (ping) request id=0x04d2, seq=0/0, ttl=127 (r
11	76.777395	192.168.1.4	192.168.1.10	ICMP	60	Echo (ping) reply id=0x04d2, seq=0/0, ttl=64 (re
12	81.982178	a6:d6:30:58:6f:f4	fc:00:02:39:3d:db	ARP	42	Who has 192.168.1.10? Tell 192.168.1.4
13	81.982675	fc:00:02:39:3d:db	a6:d6:30:58:6f:f4	ARP	60	192.168.1.10 is at fc:00:02:39:3d:db

Phase 2, Part 2, Switch to Apline 4



The screenshot shows the packet details for the first ICMP echo request. It is an Ethernet II frame with source MAC fc:00:02:39:3d:db and destination MAC a6:d6:30:58:6f:f4. The data is an Internet Control Message Protocol (ICMP) echo request.

Frame 10: 60 bytes on wire (480 bits), 60 bytes captured (480 bits) on interface -, id 0
Ethernet II, Src: fc:00:02:39:3d:db (fc:00:02:39:3d:db), Dst: a6:d6:30:58:6f:f4 (a6:d6:30:58:6f:f4)
Internet Protocol Version 4, Src: 192.168.1.10, Dst: 192.168.1.4
Internet Control Message Protocol

Figure 17: The Wireshark capture for phase 2, part 2 on connection Switch1 ↔ alpine-4.

Q.7.2 — For each packet, explain how it fits into the sequence that is generated as a result of running the packet stream you have configured on Ostinato.

Answer

So, from here on-wards the packets in Figures 16 and 17 will be referred to with their No. Specifically, these are: 65, 66 and 10, 11, 12, 13.

- 12 and 13 form part of the ARP correspondence which was autonomously initiated by apline-4 as described in the lab sheet.
- 65 and 66 are the first packets sent and these constitute the initial stage when applying the stream configuration in Ostinato. These packets form part of an ARP correspondence. ARP is used to establish whether the entity associated with the specified IP address, is also an L2 entity in the local network. Additionally, if the entity is an L2 entity it replies back with its MAC address. This procedure happens via an initial request for discovery. In the above case this is request No. 65. The device initiating ARP will broadcast an ARP packet inquiring about the MAC Address of the current holder of the specified IP. Each device on the local network will receive this ARP packet. Each device will then proceed to check whether they hold the requested IP address. If they do they will send their MAC address as a uni-cast ARP packet, since the MAC address of the initiator is known. In the above case this is request No. 66.
- Finally, the ICMP Echo request, No. 10, is sent when the stream is run in Ostinato. ICMP Echo requests are meant to check whether a device with the specified IP exists and whether it can service requests. This is the case above since the sender receives a reply, No. 11.

Q.7.3 — Select the ICMP packet generated by Ostinato. Use a tabular structure, exemplified by Fig. 39, to name **all** the fields, for all the layers, in the ICMP packet. For each field name, write a prefix to indicate which layer the field pertains to, e.g., L1 for layer 1, L2 for layer 2, etc.

Answer

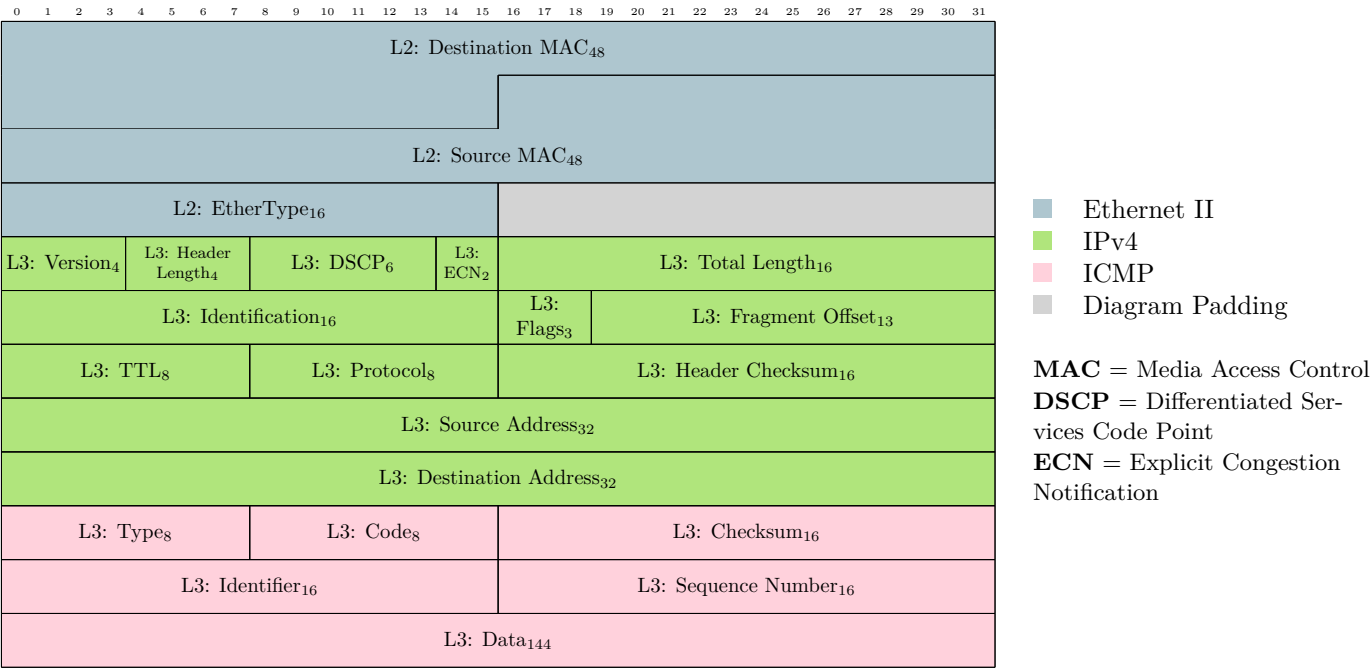


Figure 18: A diagram of the structure of an ICMP packet as provided by Wireshark.

Note: The bits marked as “Diagram Padding” in Figure 18 are *only* present to allow for a properly partitioned diagram, they are *not* present “on the wire”, *i.e.* all the fields are packed. Additionally, each field is marked with its respective bit size (as a subscript), network layer (as “LN:”) and packet type (as a colour).

Q.8 — Explain why the packets captured on Switch1 ↔ alpine-2 do not change between the point in time just before running the packet stream on Ostinato, and the point in time just after running it.

Answer

No packets are captured on connection Switch1 ↔ alpine-2 in phase 2, part 2, see Figure 15. This is because the switch as a network device keeps a mapping from ports to MAC addresses and vice-versa. This means it is capable of selectively replaying packets to the intended recipient directly. In fact, the rest of the communication is only held on connection Switch1 ↔ alpine-4, see Figure 17.

Q.9 — This concerns the dynamics of transmission using TCP. For each of the four cases (lossless, and packet loss at the rate of 1%, 3% and 5% respectively):

1. Plot a 1-s MA of the throughput, and
2. calculate the average throughput

Answer

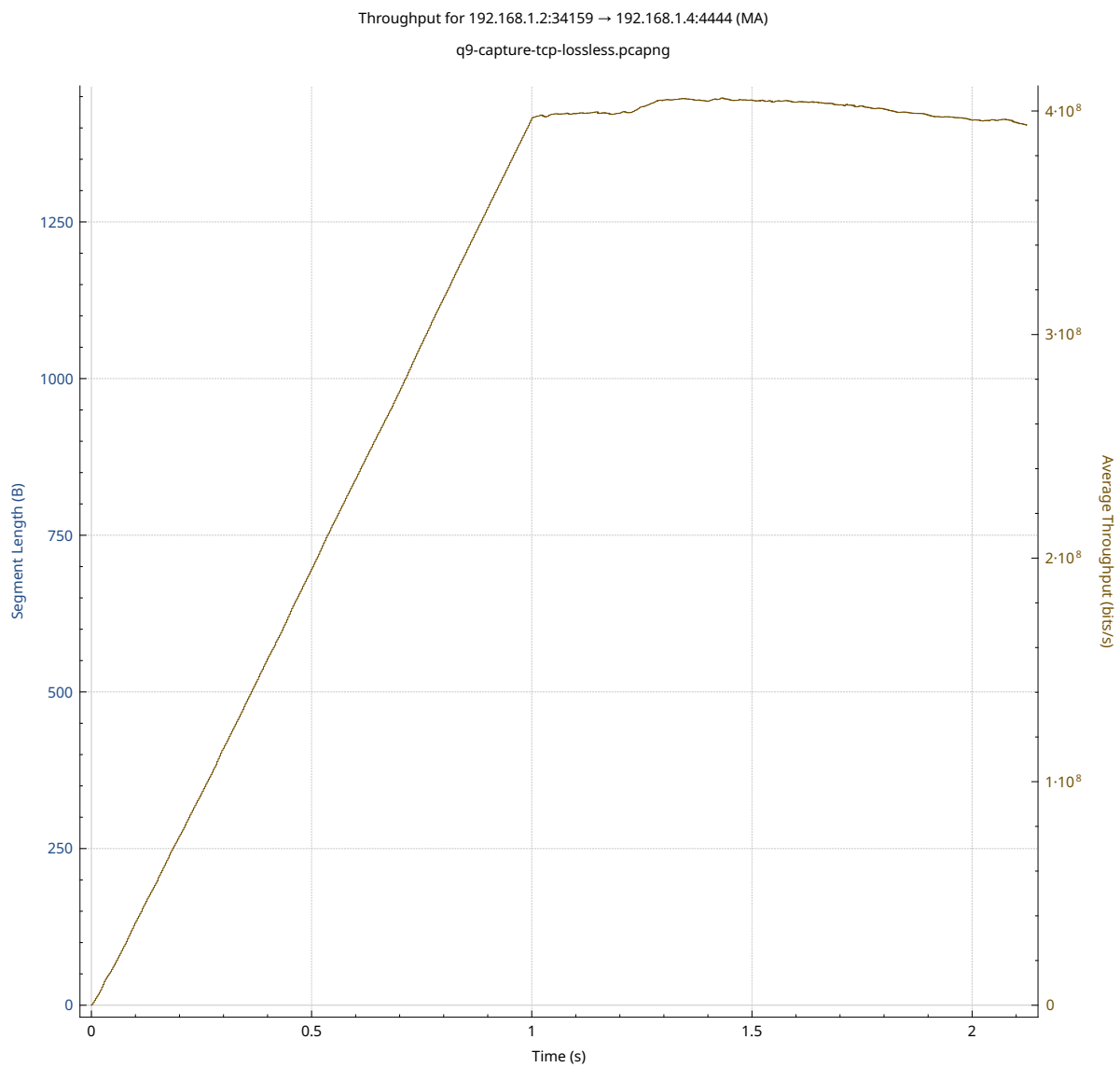


Figure 19: A plot of the 1-second MA of throughput against time (lossless).

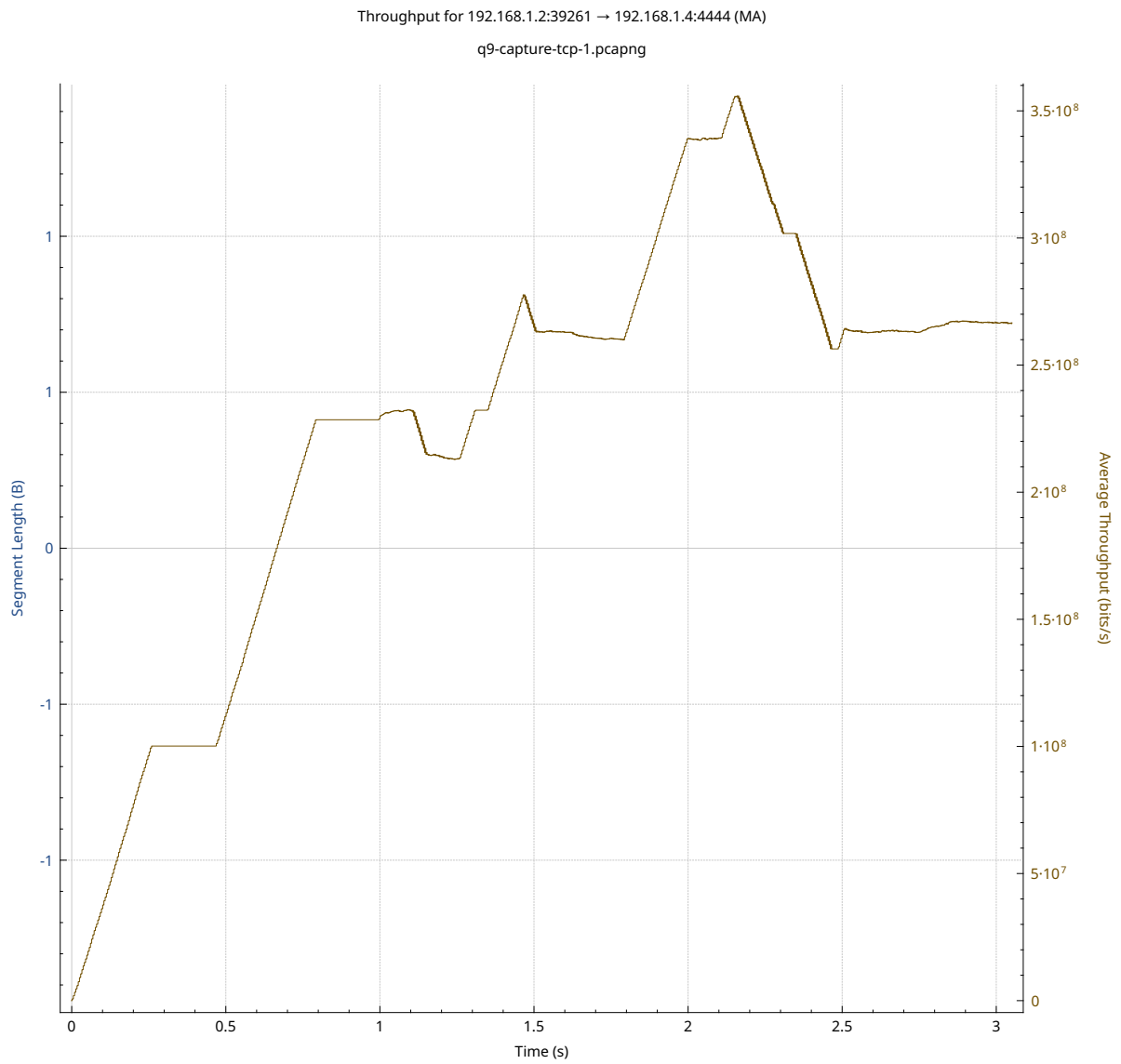


Figure 20: A plot of the 1-second MA of throughput against time (1% loss).

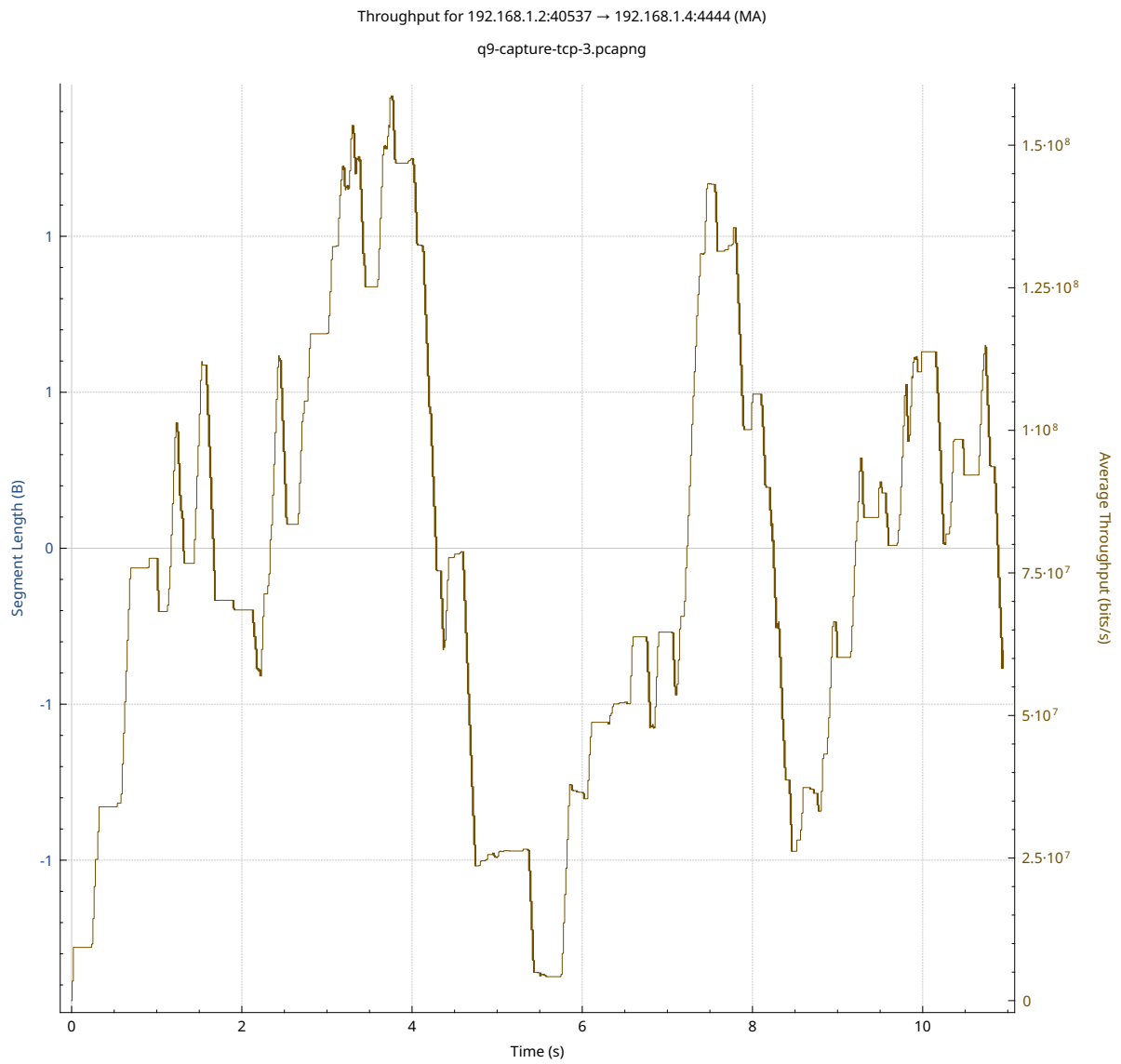


Figure 21: A plot of the 1-second MA of throughput against time (3% loss).

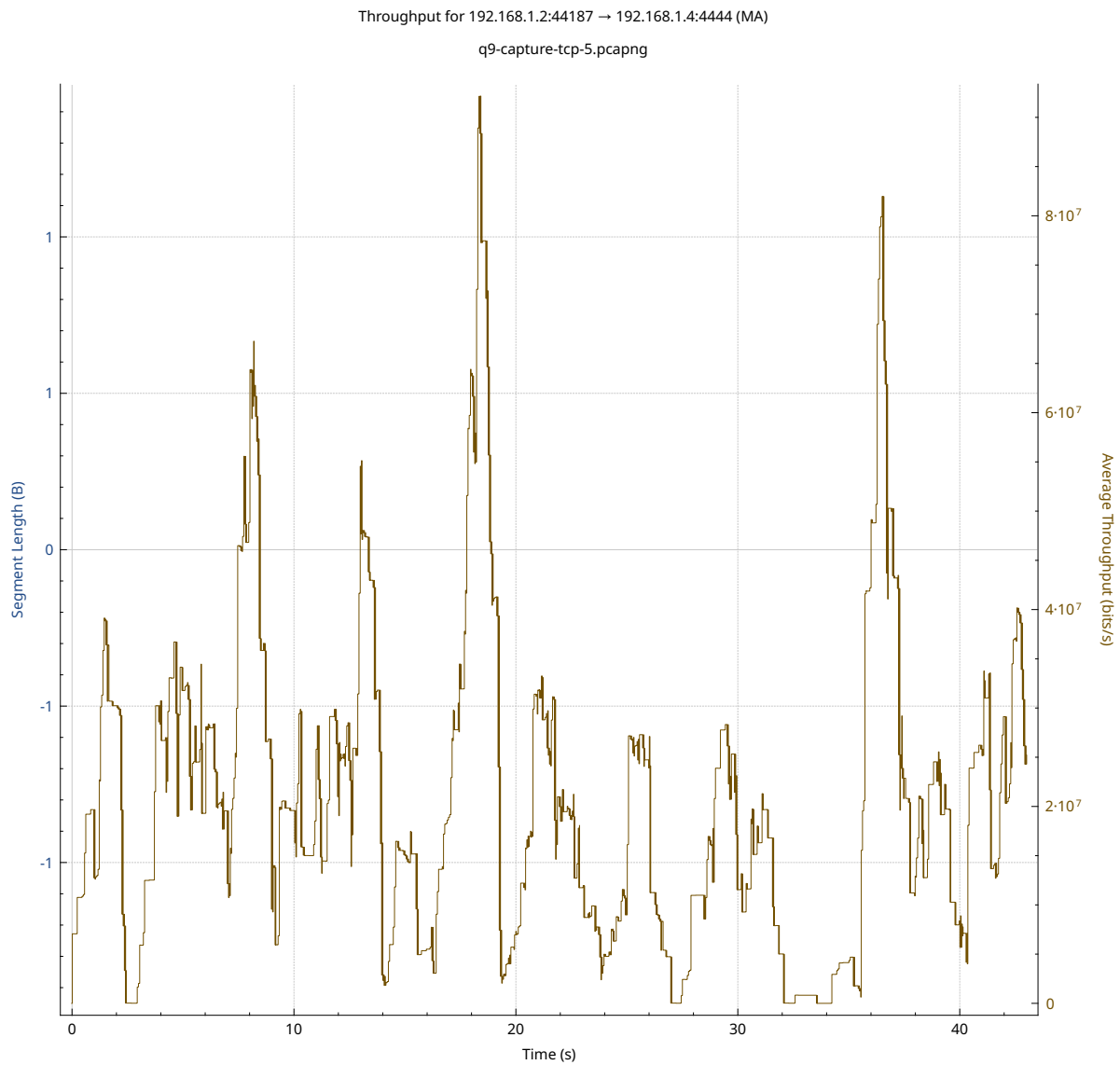


Figure 22: A plot of the 1-second MA of throughput against time (5% loss).

Figures 19, 20, 21 and 22 are the plots generated from Wireshark of a 1-second moving average (MA) of throughput against time. They were generated at different loss levels as indicated.

Listing 3: A Python script for computing a number of statistics about any Wireshark capture.

```

1 import argparse
2 import csv
3
4 def process_csv(file_path): try: with open(file_path, 'r')
5 as csv_file: reader = csv.reader(csv_file, delimiter=',',

```



```

6 quotechar='\"') data = [(float(row[1]), int(row[5])) for row
7 in [row for row in reader][1:]]
8
9 packet_count = len(data) total_size = sum((length for _,
10 length in data)) start_time = min((time for time, _ in
11 data)) end_time = max((time for time, _ in data)) diff_time
12 = end_time - start_time average_throughput =
13 total_size/diff_time file_size = 100*1024*1024
14 average_effective_throughput = file_size / diff_time
15
16 print( f"""General Info -> Packet Count: {packet_count}
17 Total Size (in Bytes): {total_size} Start Time (in
18 Seconds): {start_time} End Time (in Seconds): {end_time}
19 Difference (in Seconds): {diff_time} Average Throughput (in
20 Bytes/Second): {total_size}/{diff_time} =
21 {average_throughput}
22
23 Conversions (Throughput) -> Average Throughput (in
24 MegaBits/Second): {(average_throughput * 8) / (1000 ** 2)}
25 Average Throughput (in MegaBytes/Second):
26 {average_throughput / (1000 ** 2)}
27
28 Effective Throughput (For 100MibiByte File) -> Average
29 Effective Throughput (in Bytes/Second):
30 {file_size}/{diff_time} = {average_effective_throughput}
31
32 Conversions (Effective Throughput)-> Average Effective
33 Throughput (in MegaBits/Second):
34 {(average_effective_throughput * 8) / (1000 ** 2)} Average
35 Effective Throughput (in MegaBytes/Second):
36 {average_effective_throughput / (1000 ** 2)}""")
37
38 except FileNotFoundError: print(f"Error: File '{file_path}'
39 not found.") except Exception as e: print(f"An error
40 occurred: {e}")
41
42 def main(): parser =
43 argparse.ArgumentParser(description="Process a CSV file.")
44 parser.add_argument('file', metavar='FILE', help='Path to
45 the CSV file')
46
47 args = parser.parse_args() process_csv(args.file)
48
49 if __name__ == "__main__": main()

```

Listing 3, is Python script used for computing a number of metrics about any Wireshark capture which has been converted into a CSV.

Specifically, the captures associated with the above four figures were converted into CSV files. Then those CSV files were processed using the above script.

```

tex-files/data on ⌋ main [!] via 🐍 v3.11.6
> python calc-total-size.py q9-tcp-capture-lossless.csv
General Info ->
Packet Count: 108322
Total Size (in Bytes): 112014676
Start Time (in Seconds): 54.876607
End Time (in Seconds): 57.042749
Difference (in Seconds): 2.1661420000000007
Average Throughput (in Bytes/Second):  $112014676 / 2.1661420000000007 = 51711603.39442196$ 

Conversions (Throughput) ->
Average Throughput (in MegaBits/Second): 413.69282715537565
Average Throughput (in MegaBytes/Second): 51.71160339442196

Effective Throughput (For 100MibiByte File) ->
Average Effective Throughput (in Bytes/Second):  $104857600 / 2.1661420000000007 = 48407537.455993176$ 

Conversions (Effective Throughput)->
Average Effective Throughput (in MegaBits/Second): 387.26029964794543
Average Effective Throughput (in MegaBytes/Second): 48.40753745599318

```

Figure 23: The statistics generated by the Python script for lossless transfer.

```

tex-files/data on ⌋ main [!] via 🐍 v3.11.6
> python calc-total-size.py q9-tcp-capture-loss1.csv
General Info ->
Packet Count: 110868
Total Size (in Bytes): 114346616
Start Time (in Seconds): 4.025614
End Time (in Seconds): 7.075883
Difference (in Seconds): 3.050269
Average Throughput (in Bytes/Second):  $114346616 / 3.050269 = 37487387.505823255$ 

Conversions (Throughput) ->
Average Throughput (in MegaBits/Second): 299.89910004658606
Average Throughput (in MegaBytes/Second): 37.48738750582326

Effective Throughput (For 100MibiByte File) ->
Average Effective Throughput (in Bytes/Second):  $104857600 / 3.050269 = 34376509.088214844$ 

Conversions (Effective Throughput)->
Average Effective Throughput (in MegaBits/Second): 275.01207270571877
Average Effective Throughput (in MegaBytes/Second): 34.376509088214846

```

Figure 24: The statistics generated by the Python script for 1% loss transfer.

```

tex-files/data on 1 main [!] via 3 v3.11.6
> python calc-total-size.py q9-tcp-capture-loss3.csv
General Info ->
Packet Count: 113274
Total Size (in Bytes): 116966900
Start Time (in Seconds): 19.659794
End Time (in Seconds): 30.652448
Difference (in Seconds): 10.992653999999998
Average Throughput (in Bytes/Second): 116966900/10.992653999999998 = 10640460.438398227

Conversions (Throughput) ->
Average Throughput (in MegaBits/Second): 85.12368350718582
Average Throughput (in MegaBytes/Second): 10.640460438398227

Effective Throughput (For 100MibiByte File) ->
Average Effective Throughput (in Bytes/Second): 104857600/10.992653999999998 = 9538879.327958472

Conversions (Effective Throughput)->
Average Effective Throughput (in MegaBits/Second): 76.31103462366778
Average Effective Throughput (in MegaBytes/Second): 9.538879327958472

```

Figure 25: The statistics generated by the Python script for 3% loss transfer.

```

tex-files/data on 1 main [!] via 3 v3.11.6
> python calc-total-size.py q9-tcp-capture-loss5.csv
General Info ->
Packet Count: 116390
Total Size (in Bytes): 119726256
Start Time (in Seconds): 14.171428
End Time (in Seconds): 57.169954
Difference (in Seconds): 42.998526
Average Throughput (in Bytes/Second): 119726256/42.998526 = 2784426.982450515

Conversions (Throughput) ->
Average Throughput (in MegaBits/Second): 22.27541585960412
Average Throughput (in MegaBytes/Second): 2.784426982450515

Effective Throughput (For 100MibiByte File) ->
Average Effective Throughput (in Bytes/Second): 104857600/42.998526 = 2438632.431260551

Conversions (Effective Throughput)->
Average Effective Throughput (in MegaBits/Second): 19.509059450084408
Average Effective Throughput (in MegaBytes/Second): 2.438632431260551

```

Figure 26: The statistics generated by the Python script for 5% loss transfer.

Note: The Python script and CSV files used are present in the `data` directory.

From the above figures the following results can be read.

Table 3: The throughput results generated by the Python script (to two decimal places).

	Throughput (MB/s)	Effective Throughput (MB/s)
Lossless	51.71	48.41
1% Loss	37.49	34.38
3% Loss	10.64	9.54
5% Loss	2.78	2.44

Note: A distinction was made between **throughput** and **effective throughput**. Throughput gives the true throughput of the wire *i.e.* how many raw bytes can be transferred per second over the wire. The equation is given as follows:

$$\text{Throughput} = \frac{\text{Total Data (incl. Protocol Overhead)}}{\text{End Time} - \text{Start Time}}$$

On the other hand, effective throughput gives the usable or useful throughput of the wire *i.e.* how many bytes of interest to the user (*e.g.* a file) can be transferred per second over the wire *not* in factoring all the bytes used by the underlying transport protocol.

$$\text{Effective Throughput} = \frac{\text{Total Data (excl. Protocol Overhead)}}{\text{End Time} - \text{Start Time}}$$

Q.10 — Consider the packet sequence pertaining to the lossless case.

1. List the flags (within square brackets) pertaining to the first three packets.
2. What part of connection establishment do the first three packets pertain to?
3. List the sequence (Seq=) and acknowledgement (Ack=) numbers pertaining to the first three packets.
4. Identify the maximum segment size which the two parties in the connection state.
5. Identify the range of packets involved in the data transfer phase.
6. Identify the packet(s) involved in the connection teardown phase.

Show screenshots that allow a reader to validate your answers.

Answer

No.	Time	Source	Destination	Protocol	Length	Info
10	54.876607	192.168.1.2	192.168.1.4	TCP	74	34159 → 4444 [SYN] Seq=0 Win=32120 Len=0 MSS=1460 SACK_PERM TSval=709005049 TSecr=0 WS=128
11	54.876694	192.168.1.4	192.168.1.2	TCP	74	4444 → 34159 [SYN, ACK] Seq=0 Ack=1 Win=31856 Len=0 MSS=1460 SACK_PERM TSval=832741141 TSecr=709005049 WS=128
12	54.876951	192.168.1.2	192.168.1.4	TCP	66	34159 → 4444 [ACK] Seq=1 Ack=1 Win=32128 Len=0 TSval=709005049 TSecr=832741141

Figure 27: Screenshot of the first three TCP packets from the lossless capture.

1. SYN for the first packet, SYN and ACK for the second packet and finally, ACK for the third packet.

2. These three packets form part of the Three-Way Handshake which establishes a reliable connection between the sender and receiver.
3. Seq = 0 for the first packet, Seq = 0 and Ack = 1 for the second packet and finally, Seq = 1 and Ack = 1 for the third packet.
4. The maximum segment size (MSS) is 1460 bytes, see the second packet.

108329	57.000153	192.168.1.2	192.168.1.4	TCP	1242	34159 → 4444	[FIN, PSH, ACK]	Seq=104856425	Ack=1	Win=32128	Len=1176	TSval=709007170	TSecr=832743262
108330	57.000176	192.168.1.4	192.168.1.2	TCP	66	4444 → 34159	[ACK]	Seq=1	Ack=104856425	Win=1150720	Len=0	TSval=832743265	TSecr=709007170
108331	57.042749	192.168.1.4	192.168.1.2	TCP	66	4444 → 34159	[ACK]	Seq=1	Ack=104857602	Win=1150720	Len=0	TSval=832743307	TSecr=709007170

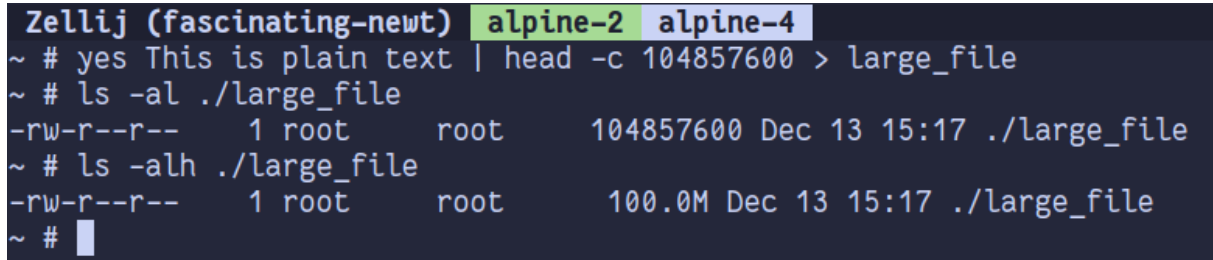
Figure 28: Screenshot of the last three TCP packets from the lossless capture.

5. From Figures 27 & 28 the range of the packets is 10 to 108331.
6. The packets that form part of the teardown are those in Figure 28.

Q.11 — Consider the packet sequence pertaining to the 5% packet loss case. List `received_file` on alpine-4 and compare its size with that of `large_file`.

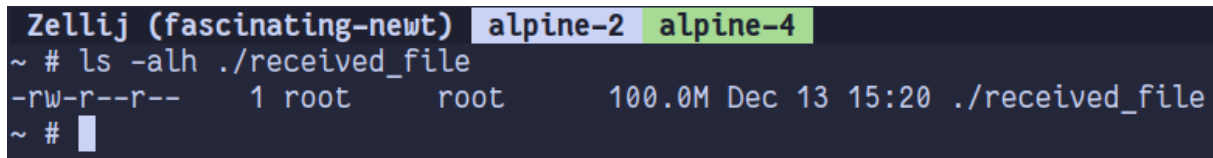
Show screenshots that allow a reader to validate your answers.

Answer

A terminal window titled 'Zellij (fascinating-newt)' with tabs for 'alpine-2' and 'alpine-4'. The 'alpine-2' tab is active. The user enters the command 'yes This is plain text | head -c 104857600 > large_file'. Then, they run 'ls -al ./large_file', showing a file of size 104857600. Finally, they run 'ls -alh ./large_file', showing the file size as 100.0M.

```
Zellij (fascinating-newt) alpine-2 alpine-4
~ # yes This is plain text | head -c 104857600 > large_file
~ # ls -al ./large_file
-rw-r--r--  1 root    root      104857600 Dec 13 15:17 ./large_file
~ # ls -alh ./large_file
-rw-r--r--  1 root    root      100.0M Dec 13 15:17 ./large_file
~ #
```

Figure 29: Large file generation on alpine-2.

A terminal window titled 'Zellij (fascinating-newt)' with tabs for 'alpine-2' and 'alpine-4'. The 'alpine-4' tab is active. The user runs 'ls -alh ./received_file', showing a file of size 100.0M.

```
Zellij (fascinating-newt) alpine-2 alpine-4
~ # ls -alh ./received_file
-rw-r--r--  1 root    root      100.0M Dec 13 15:20 ./received_file
~ #
```

Figure 30: Listing received large file on alpine-4.

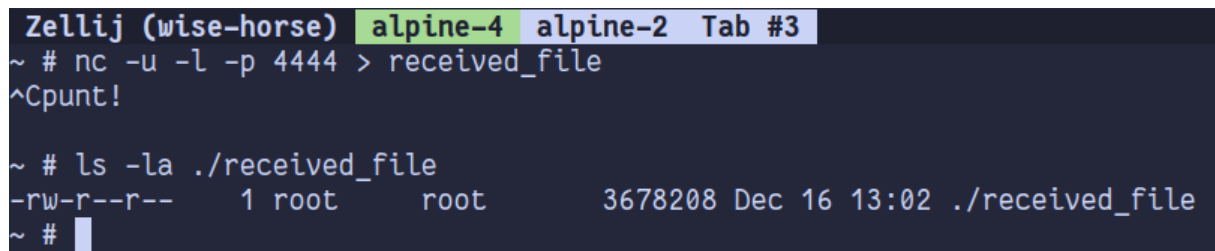
As can be seen from Figures 29 & 30 the file size remained in the same. This is because TCP is a reliable data transport protocol *i.e.* it provides a zero-tolerance policy for data loss.

Q.12 — Consider the packet sequence pertaining to the lossless case.

1. List `received_file` on `alpine-4` and compare its size with that of `large_file`.
2. Go to File->Export Packet Dissections, export the captured packets as CSV and inspect the CSV file in Excel. How does the number of octets sent (as you determine from the CSV file) compare with the size of `received_file`?
3. Inspect the first few packets in the UDP window and identify the length of the UDP datagram (“Len”). How does this compare with the Ethernet frame’s MTU of 1500? Explain any difference you observe.

Show screenshots that allow a reader to validate your answers.

Answer



```
Zellij (wise-horse) alpine-4 alpine-2 Tab #3
~ # nc -u -l -p 4444 > received_file
^Cpunt!

~ # ls -la ./received_file
-rw-r--r--  1 root    root      3678208 Dec 16 13:02 ./received_file
~ #
```

Figure 31: Listing received large file on `alpine-4` for Question 12.1.

The `received_file` in `alpine-4` has a size of 3678208 bytes. This is roughly $100 \times \frac{3678208}{104857600} \approx 3.5\%$ of the original size of `large_file`.

Note: This seems to be a consequence of using GNS3 with two Docker containers. It might be related to the fact that Netcat does not allow enough time for the packets to be sent. Instead it tries to send all packets at once which causes Linux in the Docker containers or the Docker containers themselves to drop a significant amount of the packets generated by Netcat.

```

network-cce2414/lab/experiment
> nc -u -q 10 127.0.0.1 4444 < large_file

network-cce2414/lab/experiment took 10s
>

network-cce2414/lab/experiment
> ls -al
total 102400
drwxr-xr-x 1 juan juan      20 Dec 16 14:26 .
drwxr-xr-x 1 juan juan    148 Dec 14 18:16 ..
-rw-r--r-- 1 juan juan 104857600 Dec 14 18:17 large_file

network-cce2414/lab/experiment
> nc -u -l -p 4444 > received_file
^C

network-cce2414/lab/experiment took 17s
> ls -al
total 182576
drwxr-xr-x 1 juan juan      46 Dec 16 14:27 .
drwxr-xr-x 1 juan juan    148 Dec 14 18:16 ..
-rw-r--r-- 1 juan juan 104857600 Dec 14 18:17 large_file
-rw-r--r-- 1 juan juan 82100224 Dec 16 14:27 received_file

network-cce2414/lab/experiment
>

```

Figure 32: Using Netcat in UDP mode for local file transfer using the Loopback device.

As can be seen from Figure 32 the size of `received_file` is 82100224 bytes. This is $100 \times \frac{82100224}{104857600} \approx 78.3\%$ of the original file size. A much better and frankly more expected outcome. Additionally, there is also the possibility that BusyBox Netcat has different behavior from OpenBSD Netcat. Also extra care was taken to ensure that no mistake was made and that artificial packet loss was reset to 0%.

Please zoom in onto the Figure 32 for improved clarity.

For the second part of the question again a Python script similar to Listing 3 is used. The only difference is that the size of UDP payloads is assumed to be exactly 8192 bytes.

Listing 4: A Python script for computing a number the total size of a number of maximum size (8192 bytes) UDP packets.

```

1 import argparse
2 import csv
3
4 def process_csv(file_path): try: with open(file_path, 'r')
5 as csv_file: reader = csv.reader(csv_file, delimiter=',',
6 quotechar='"') data = [(float(row[1]), int(row[5])) for row
7 in [row for row in reader][1:]]
8
9 packet_count = len(data) total_size = 8192 * packet_count
10 start_time = min((time for time, _ in data)) end_time =
11 max((time for time, _ in data)) diff_time = end_time -
12 start_time
13
14 print( f"""General Info -> Packet Count: {packet_count}
15 Total Size (in Bytes): {total_size} Start Time (in
16 Seconds): {start_time} End Time (in Seconds): {end_time}

```



```

17 Difference (in Seconds): {diff_time}
18
19 Conversions -> Total Size (in MibiBytes): {(total_size) /
20 (1024 ** 2)}"""
21
22 except FileNotFoundError: print(f"Error: File '{file_path}'
23 not found.") except Exception as e: print(f"An error
24 occurred: {e}")
25
26 def main(): parser =
27 argparse.ArgumentParser(description="Process a CSV file.")
28 parser.add_argument('file', metavar='FILE', help='Path to
29 the CSV file')
30
31 args = parser.parse_args() process_csv(args.file)
32
33 if __name__ == "__main__": main()

```

```

tex-files/data on ♪ main [!?] via 🐍 v3.11.6
> python udp-total-size.py q12-udp-capture-lossless.csv
General Info ->
Packet Count: 847
Total Size (in Bytes): 6938624
Start Time (in Seconds): 171.118097
End Time (in Seconds): 171.245407
Difference (in Seconds): 0.127309999999999426

Conversions ->
Total Size (in MibiBytes): 6.6171875

```

Figure 33: Information about the UDP Wireshark capture (lossless).

Again, these calculations were made under the assumption that the `Len` information provided by the UDP packets should be used to identify the length of the UDP payload. From Figure 33 the total size of useful data captured is roughly 6.6 MibiBytes which is not the size of the `received_file` (3.6 MibiBytes). Hence, the only reasonable conclusion is that there has been some additional loss exactly upon receipt at alpine-4.

Finally, the `Len` UDP is field is set to 8192. This value is clearly greater than the 1500 bytes, which is the Maximum Transfer Unit (MTU) of an Ethernet frame. This is not contradicting the size restriction imposed at layer 2. UDP has not awareness of what is happening layer 3 and below. Hence, the whole UDP is actually dismantled into separate smaller chunks which respect the MTU and then these IP packets are reordered and recombined to reconstruct the UDP packet. This is also indicated to us by Wireshark, see Figure 34.

•	7	171.118022	192.168.1.2	192.168.1.4	IPv4	1514	Fragmented IP protocol (proto=UDP 17, off=0, ID=ebe8) [Reassembled in #12]
•	8	171.118034	192.168.1.2	192.168.1.4	IPv4	1514	Fragmented IP protocol (proto=UDP 17, off=1480, ID=ebe8) [Reassembled in #12]
•	9	171.118045	192.168.1.2	192.168.1.4	IPv4	1514	Fragmented IP protocol (proto=UDP 17, off=2960, ID=ebe8) [Reassembled in #12]
•	10	171.118062	192.168.1.2	192.168.1.4	IPv4	1514	Fragmented IP protocol (proto=UDP 17, off=4440, ID=ebe8) [Reassembled in #12]
•	11	171.118080	192.168.1.2	192.168.1.4	IPv4	1514	Fragmented IP protocol (proto=UDP 17, off=5920, ID=ebe8) [Reassembled in #12]
←	12	171.118097	192.168.1.2	192.168.1.4	UDP	834	38730 → 4444 Len=8192

Figure 34: A subsection of the UDP Wireshark capture demonstrating packet reassembly.

Q.13 — Explain the difference between your observations in 11 and 12.1.

Answer

The major take away from the procedures held to answer questions 11 and 12 is that TCP and UDP are very different in their approach. TCP is considered to be connection-oriented and reliable, this is because TCP ensures that data is not lost, not corrupted and totally delivered. This is even the case when the network is not stable (as simulated in question 11 with a 5% packet loss rate).

On the other hand UDP is known as a stateless and connection-less data transfer protocol. This is because UDP does not make any guarantee about the reliability of data transfer. In fact, it is possible that no the data sent over UDP actually reaches its intended destination. Additionally, UDP doesn't even attempt to check whether the recipient of the data is available to receive.

In fact, Netcat will work when started in UDP mode even when no recipient is available. However, in TCP mode Netcat will actually exit if the Three-Way Handshake for connection establishment fails.

However, there are some applications which benefit from UDP. This is because UDP is a much leaner and hence faster data transfer protocol. This is because UDP does not need to establish a connection and more importantly since it does not establish a connection it does not need to maintain said connection. This makes UDP more suited form applications with continuous streams of data generation which are tolerant of some missing data. The best examples of this are multi-player first-person shooters (type of games). This is because accuracy within the data is not required and the same data is often set multiple times.