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

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 introduces another level of complexity which I do not believe is feasible to consider here.

Also, I believe it is 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 try and provide a hopefully accurate explanantion.

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 when there is more than one device, 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 the 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.

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.

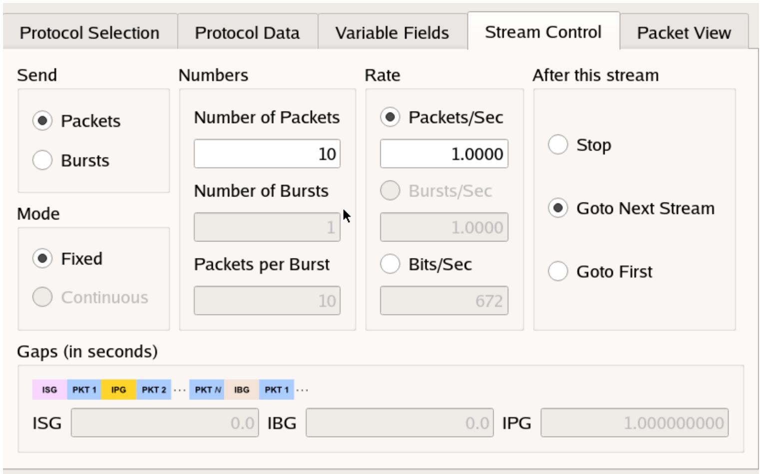


Figure 1: The stream settings in Ostinato for Question 2.1.

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 from 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

$$\frac{\begin{aligned}&(4618.638764 - 4617.638757) + (4619.638761 - 4618.638764) + (4620.638772 - 4619.638761)\\ &+ (4621.638731 - 4620.638772) + (4622.638714 - 4621.638731) + (4623.638729 - 4622.638714)\\ &+ (4624.638727 - 4623.638729) + (4625.638735 - 4624.638727) + (4626.638725 - 4625.638735)\end{aligned}}{10 - 1}$$

$$= 0.9999964444444535 \text{ seconds}$$

$$\approx 1 \text{ seconds}$$

The above computed value is the average inter-packet delay (in seconds). The approximate value of the above result is 1 second. This 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.

```
((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))/9
```

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

Answer

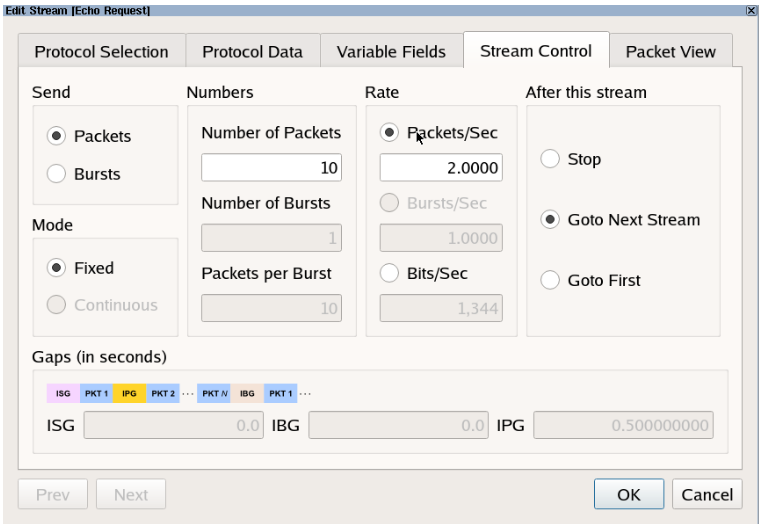


Figure 2: The stream settings in Ostinato for Question 3

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).

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

$$= 0.4999943333333335 \text{ seconds}$$

$$\approx 0.5 \text{ seconds}$$

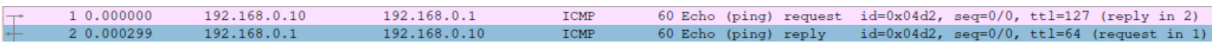
The approximate value of the above result is 0.5 seconds. This is precisely what should be expected since if two request are being sent per second (see Figure 2) that would equate to a request every half a second.

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

```
((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))/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 insception for Question 4.1.

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

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

Answer

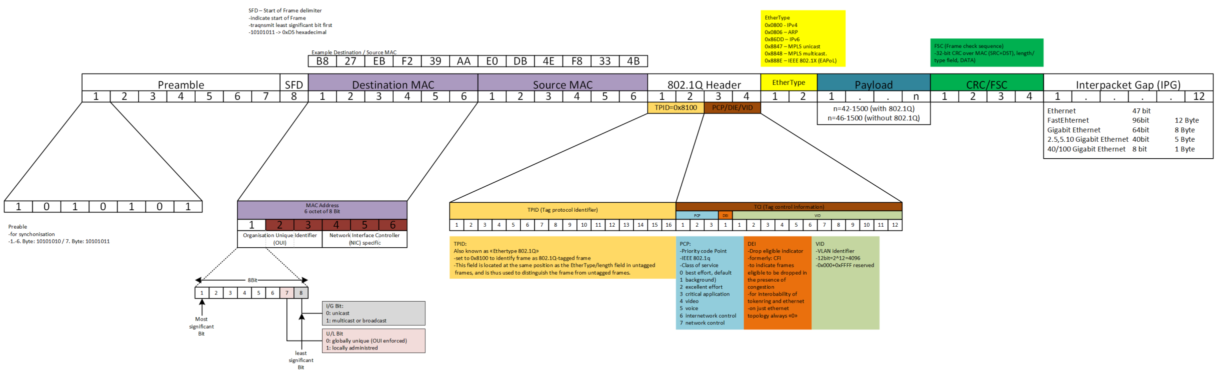


Figure 5: A diagram of the components of an Ethernet II frame.

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 as follows:

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

> **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, in both cases the sum of their lengths is identical.

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

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

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

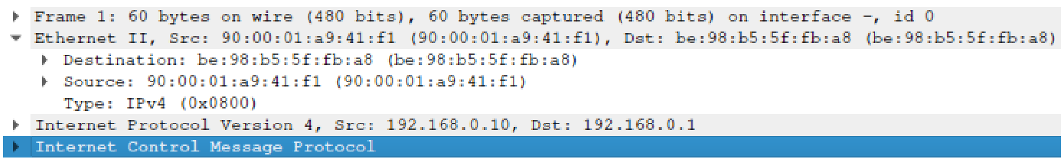


Figure 6: The Ethernet II header of the request packet under inspection for Question 4.1.

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.

To account for the 4 byte, 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 be. They constitute the Cyclic Redundancy Check (CRC) or Frame Check Sequence (FCS). In fact, according to Figure 5, the CRC field is exactly 4 bytes.

Furthermore, this conclusion is 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 a CRC fails, the network driver which performs the check will often automatically drop the invalid packet. This mean that invalid packets will never actually reach user space. Hence, providing the CRC to user space applications is often a redundant since the CRC would have already been checked to be vaild.

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

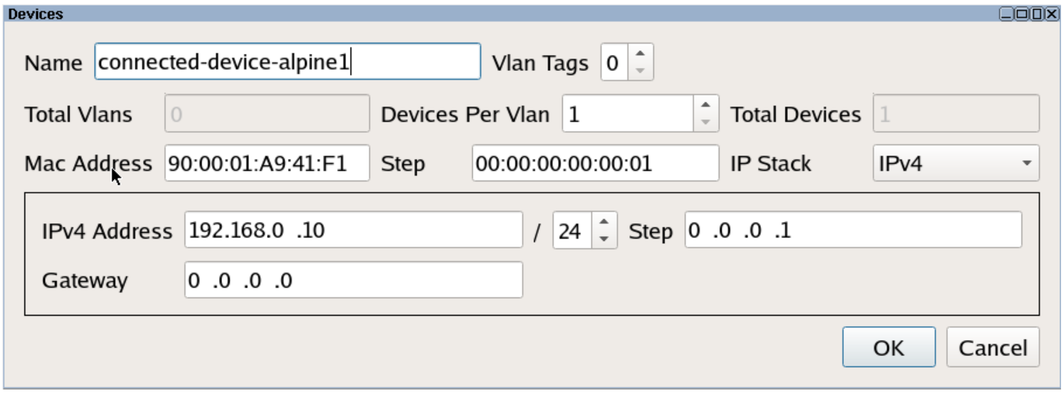


Figure 7: The device group configuration in Ostinator for Question 4.5.

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 any form of local layer 3 communication is very much dependent on ARP. This is because ARP facilitates layer 3 communication over layer 2. It does this by providing the capability to resolve *i.e.* find the MAC address of an IP address holder.

> **Note:** In this context the "local" layer 3 network of a device is all the other devices which have the same network address and are reachable via layer 2.

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.

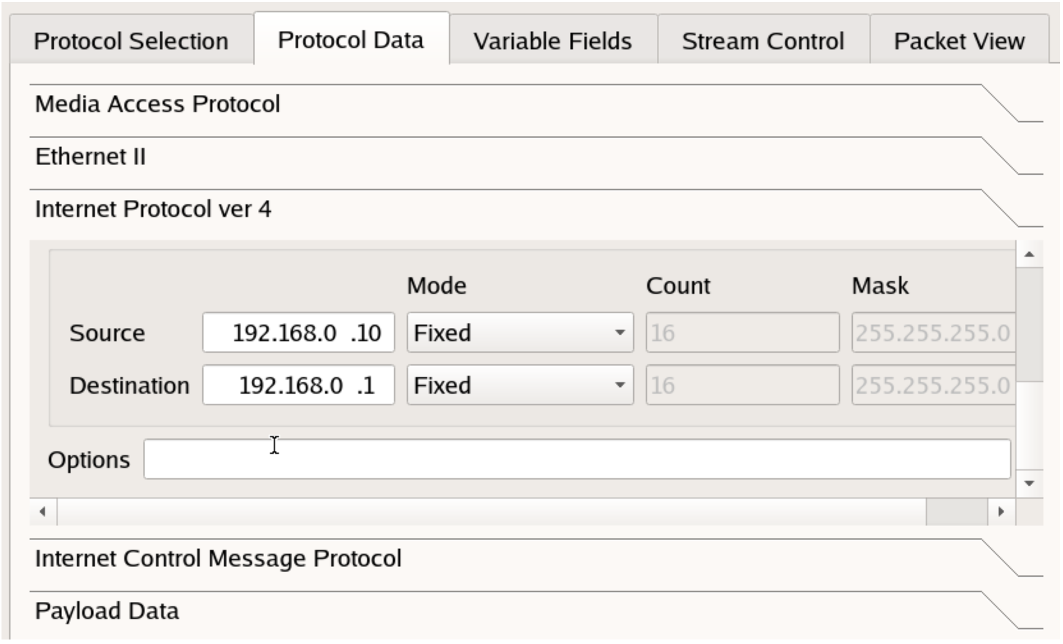


Figure 9: The stream's IPv4 configuration in Ostinato for Question 4.8.

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

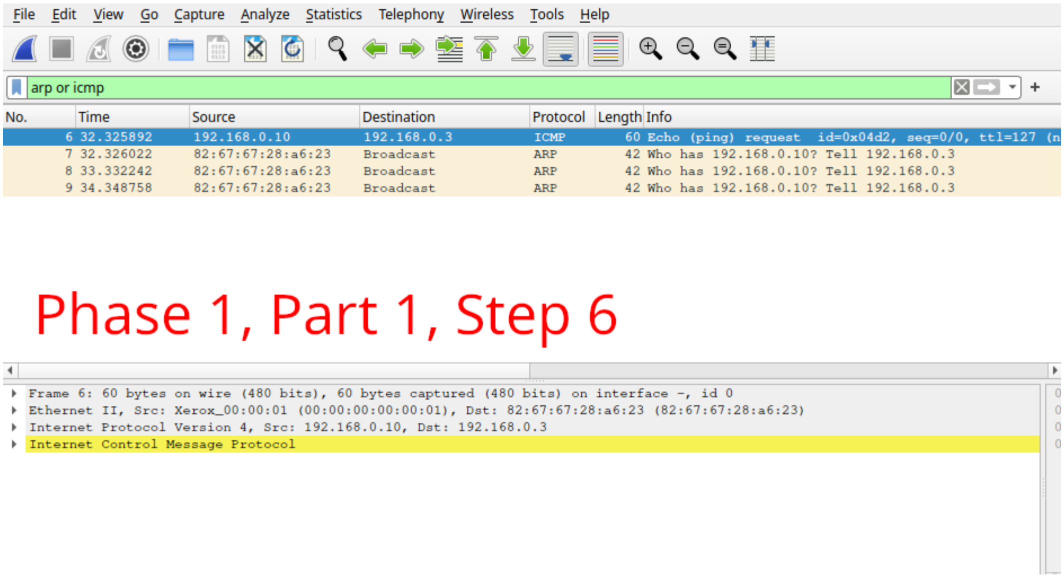


Figure 10: The Wireshark capture for phase 1, part 1, step 6.

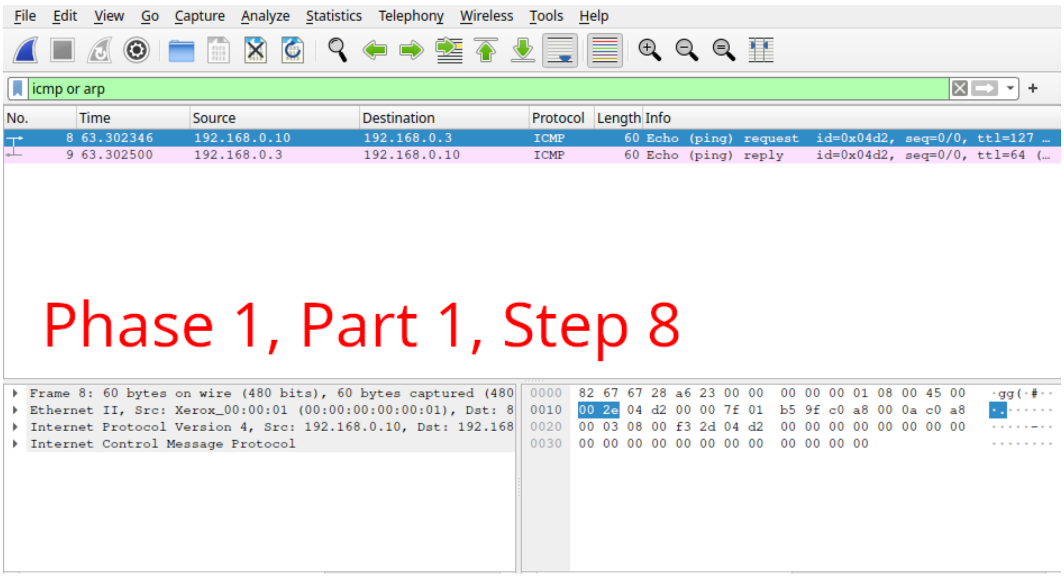


Figure 11: The Wireshark capture for phase 1, part 1, step 8.

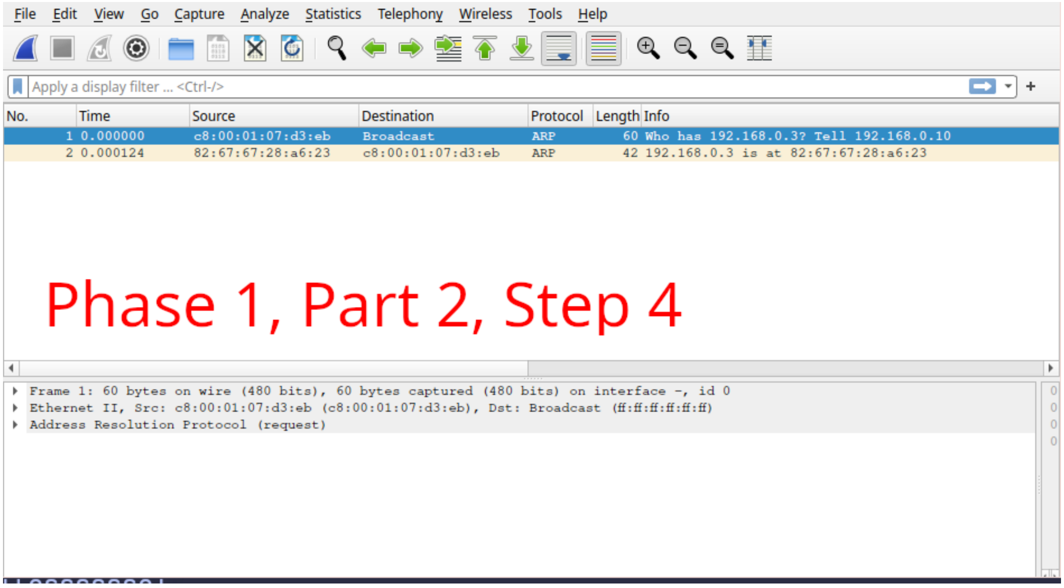


Figure 12: The Wireshark capture for phase 1, part 2, step 4.

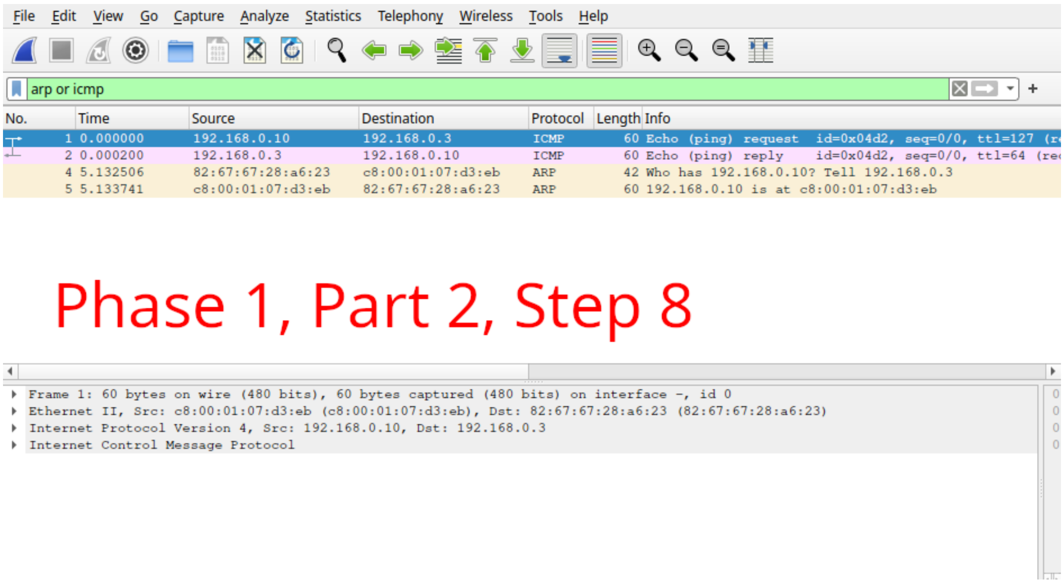


Figure 13: The Wireshark capture for phase 1, part 2, step 8.

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

As can be seen from Figure 10, the IP addresses in use are `192.168.0.10` and `192.168.0.3`. These are the IP addresses of the virtual Ostinatio devices 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 of source to/from destination map. Hubs take a broad stroke approach and replay any communication they receive into all of their other 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 ↔ apline-2 and Swtich1 ↔ apline-4 were monitored.

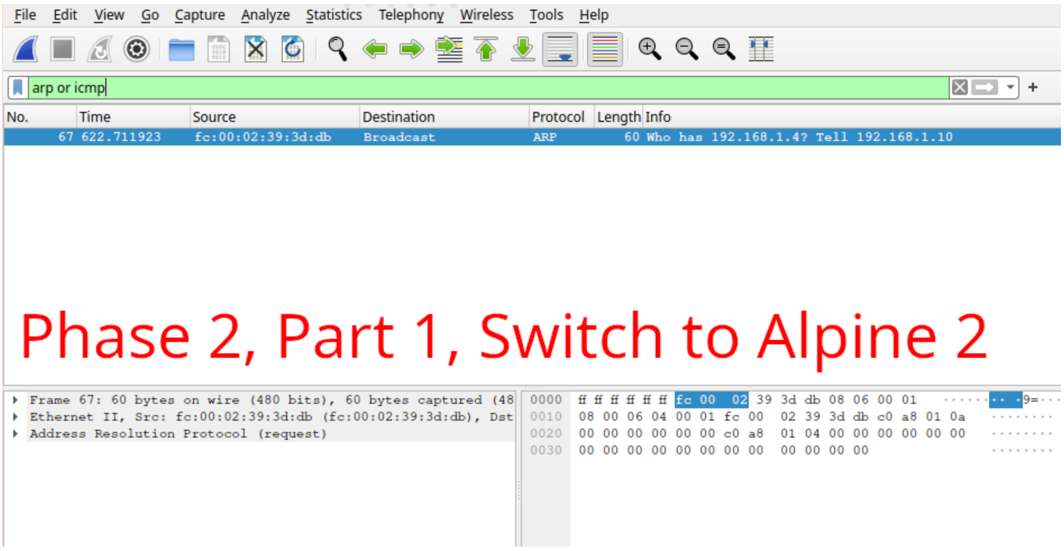


Figure 14: The Wireshark capture for phase 2, part 1 on connection Switch1 ↔ alpine-2.

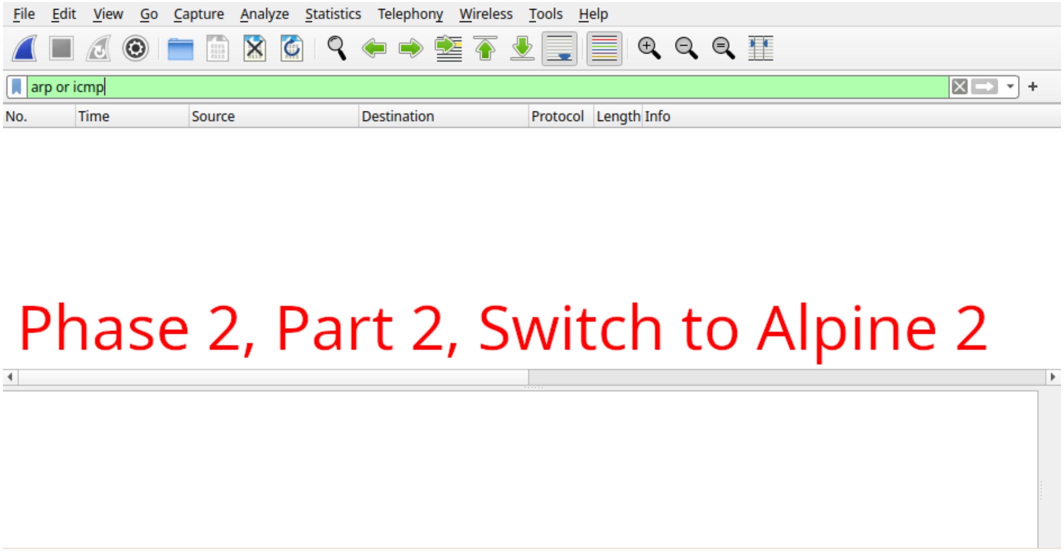


Figure 15: The Wireshark capture for phase 2, part 2 on connection Switch1 ↔ alpine-2.

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.

> **Note:** The documented difference is not very meaningful. This is because Ostinato provides the facility to manually specify the MAC addresses of its virtual devices. Hence, with a little bit of care to ensure no MAC address collisions, the same result as Fig. 37 can be achieved.

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

Answer

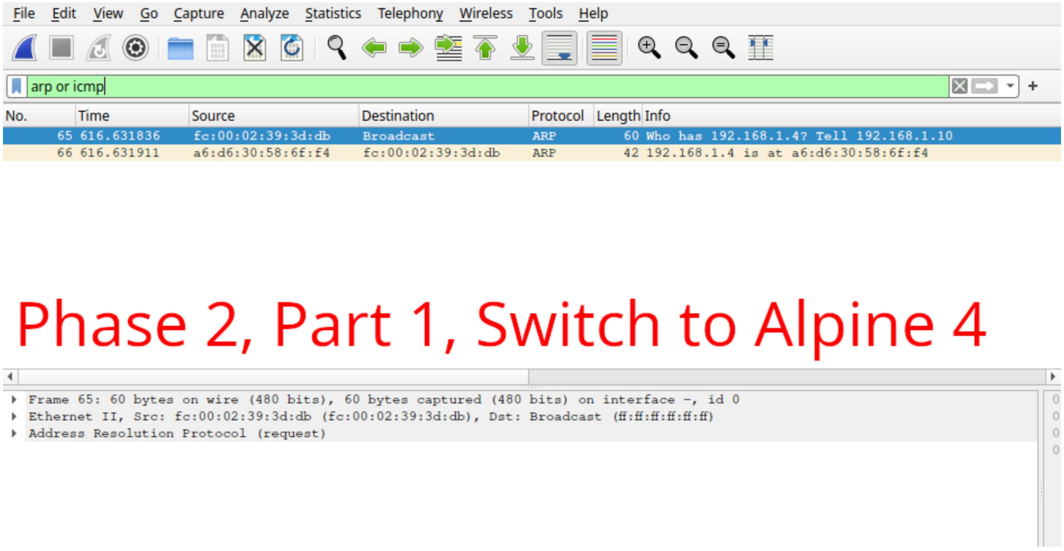


Figure 16: The Wireshark capture for phase 2, part 1 on connection Switch1 ↔ alpine-4.

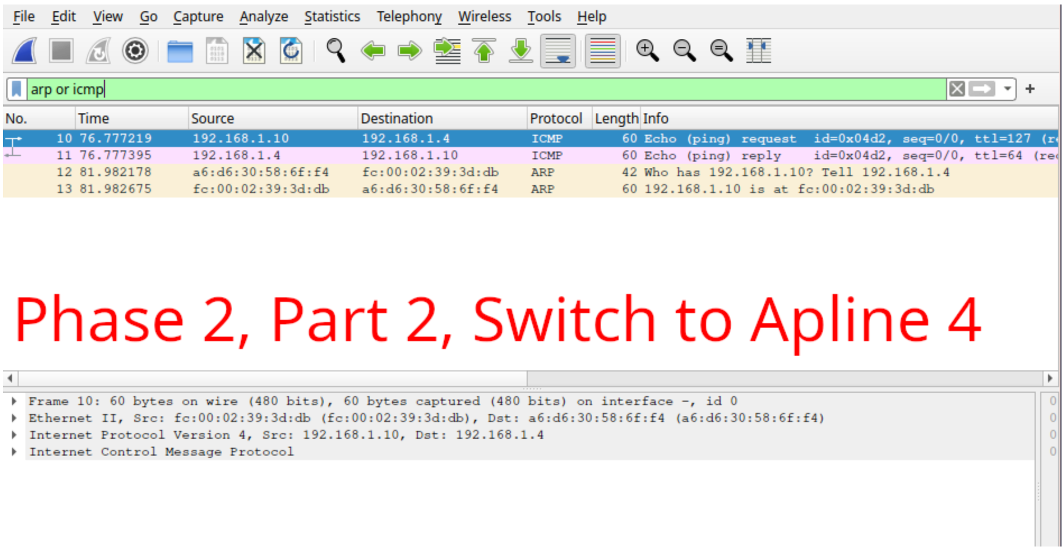


Figure 17: The Wireshark capture for phase 2, part 2 on connection Switch1 ↔ alpine-4.

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

From here on-wards the packets in Figures 16 and 17 will be refered to with there number (labelled 'No.' in the Figures). Specifically, these are: 65, 66 and 10, 11, 12, 13.

- No. 65 and No. 66 constitute the first ARP correspondence initiated by applying the stream configuration in Ostinato. 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 request is 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 reply No. 66.
- No. 10 and No. 11 are the ICMP Echo request/reply pair The request (No. 10), is sent when the configured Ostinato stream is run. The reply (No. 11) is sent by the receiver of the request to inform the initiating device (in this case an Ostinato virtual device) that it is available and ready to serve.

  > **Note:** Pings (ICMP Echo Requests) are meant to check whether a device with a specified IP address exists and whether it can service requests. This can be observed in Figure 17.

- No. 12 and No. 13 constitute the ARP correspondence which was autonomously initiated by apline-4 as described in the lab sheet.

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

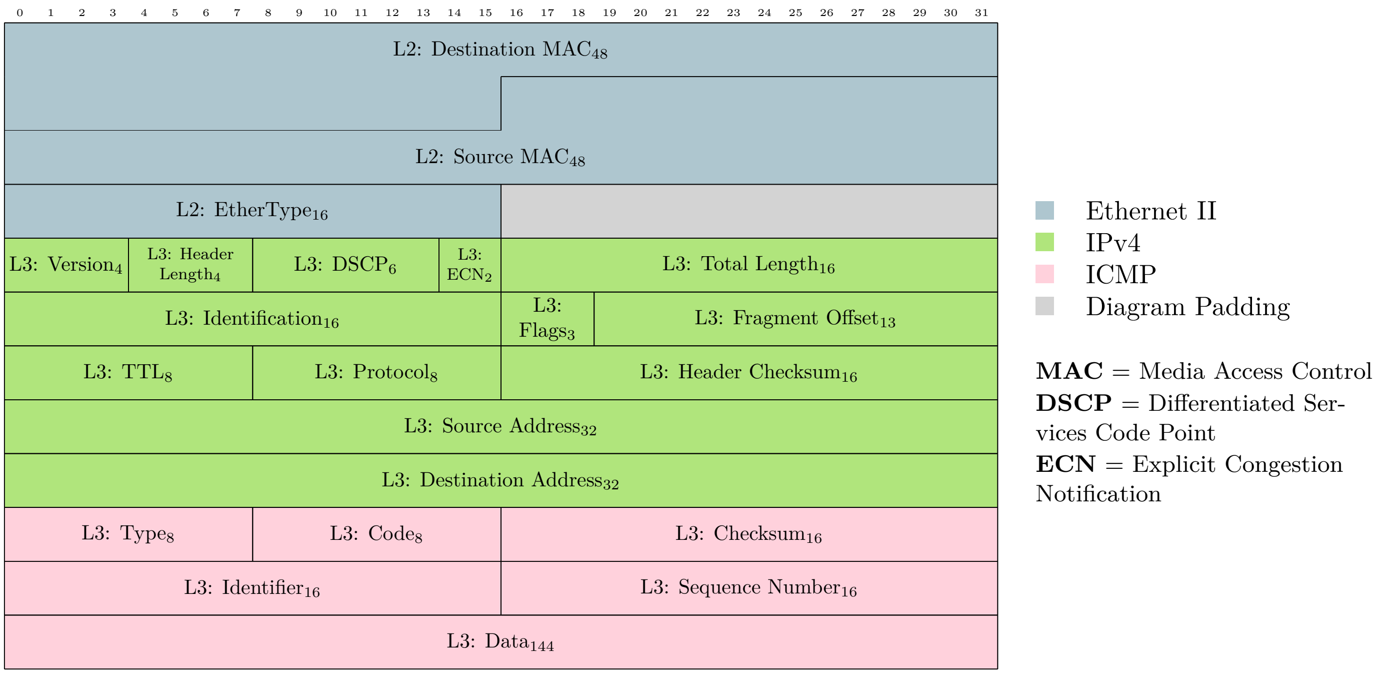


Figure 18: A diagram of the structure of an ICMP packet as provided by Wireshark.

> **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 their intended recipients 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

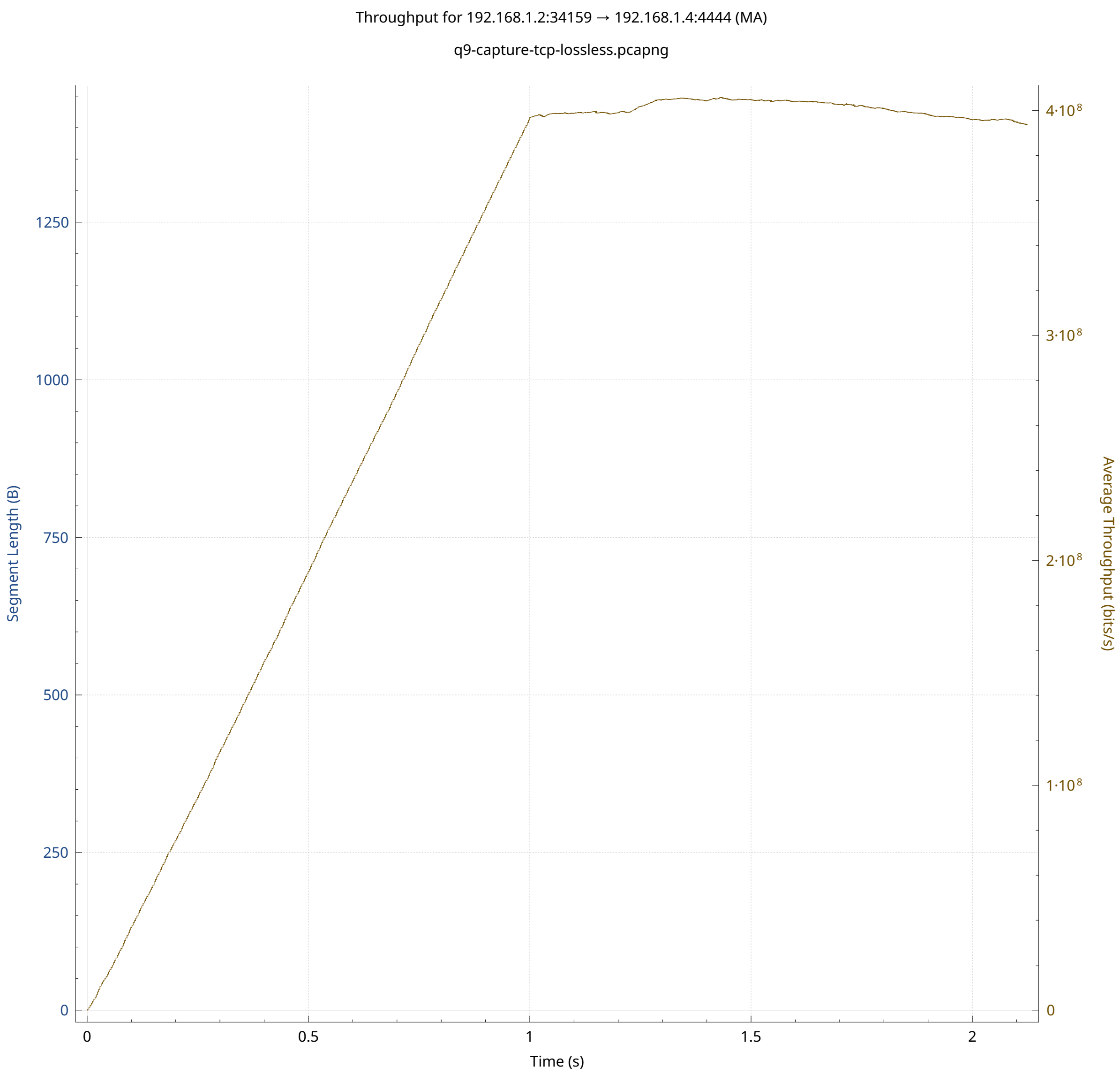


Figure 19: A plot of the 1-second MA of throughput against time (lossless).

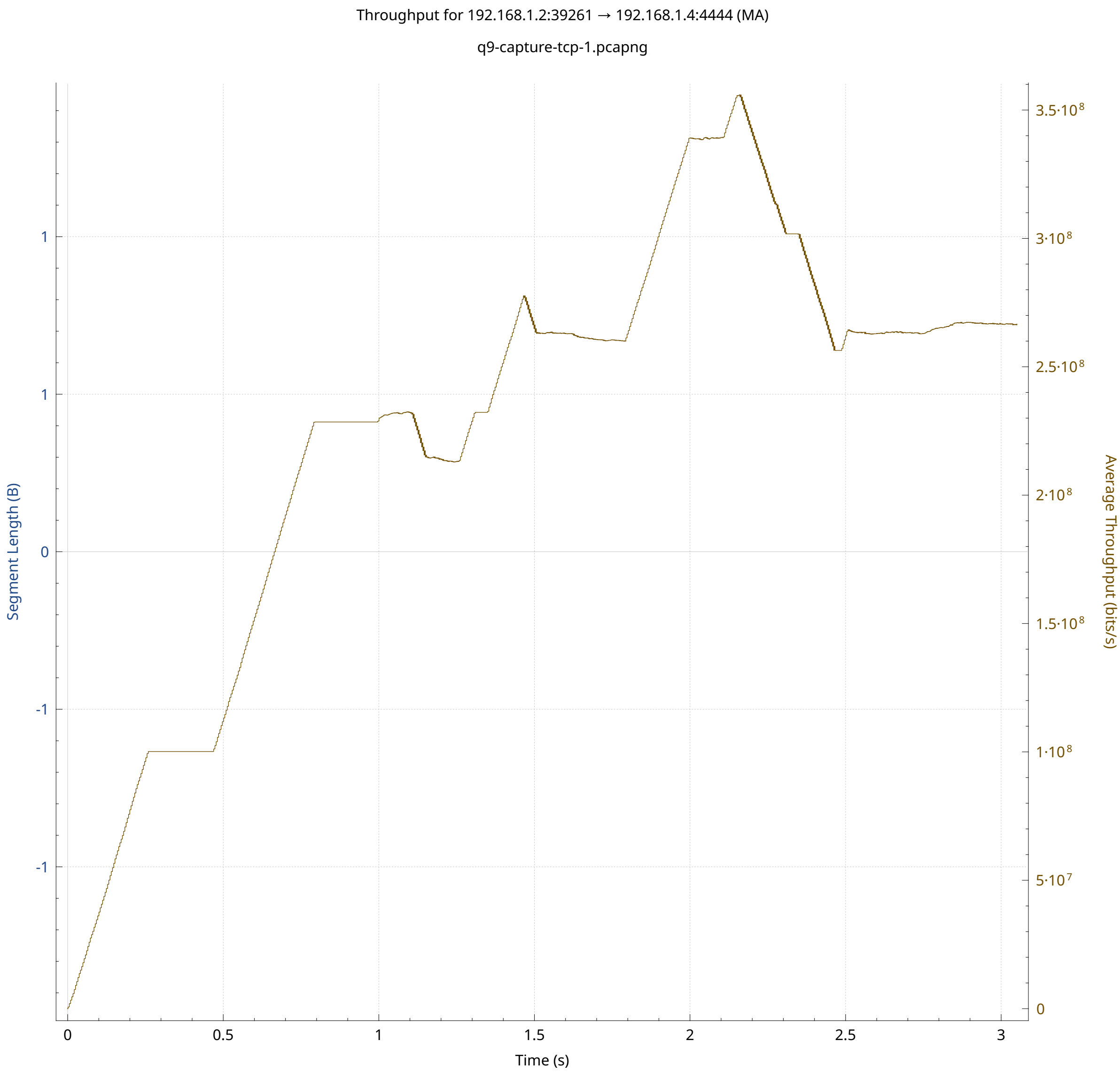


Figure 20: A plot of the 1-second MA of throughput against time (1% loss).

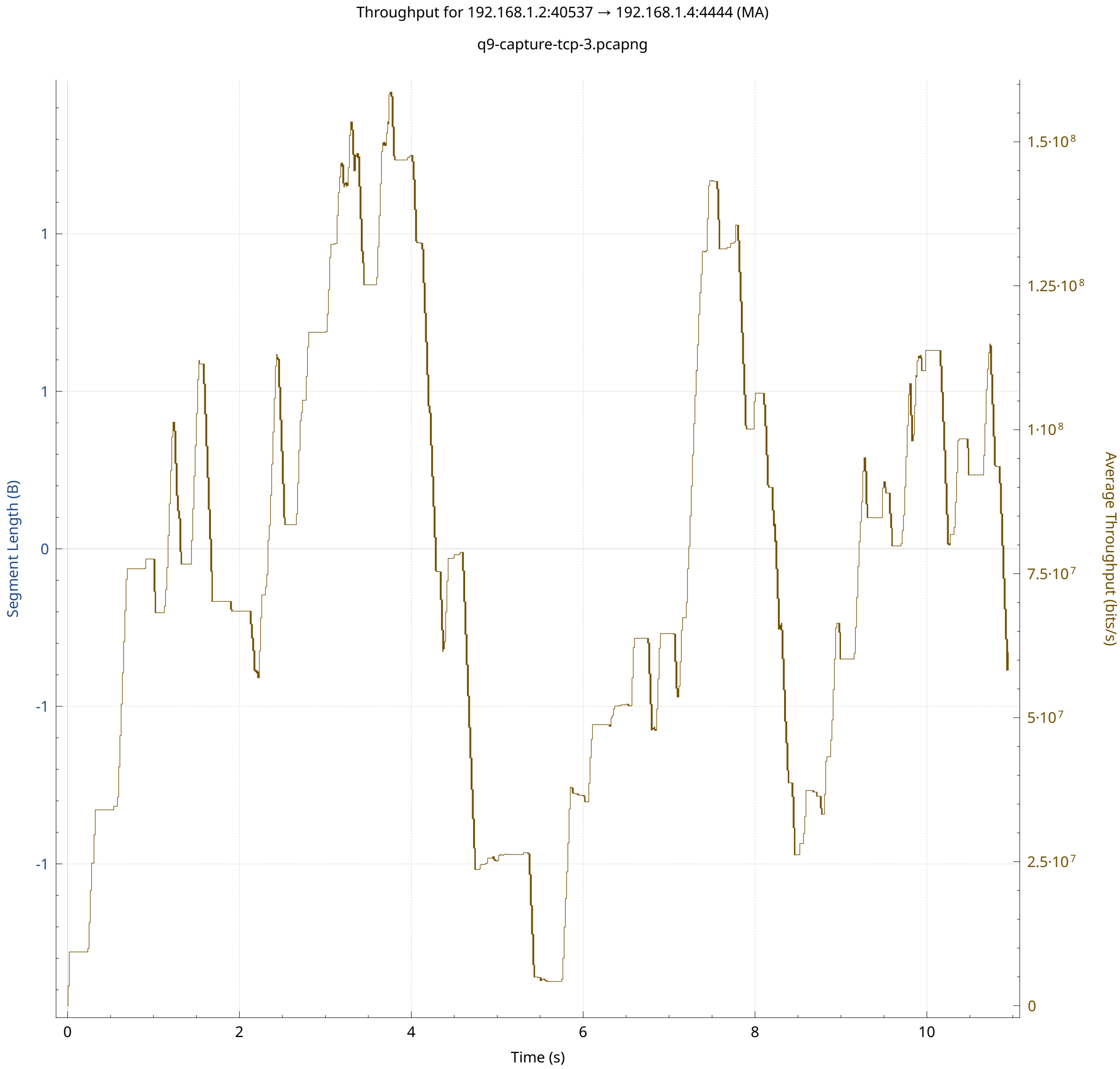


Figure 21: A plot of the 1-second MA of throughput against time (3% loss).

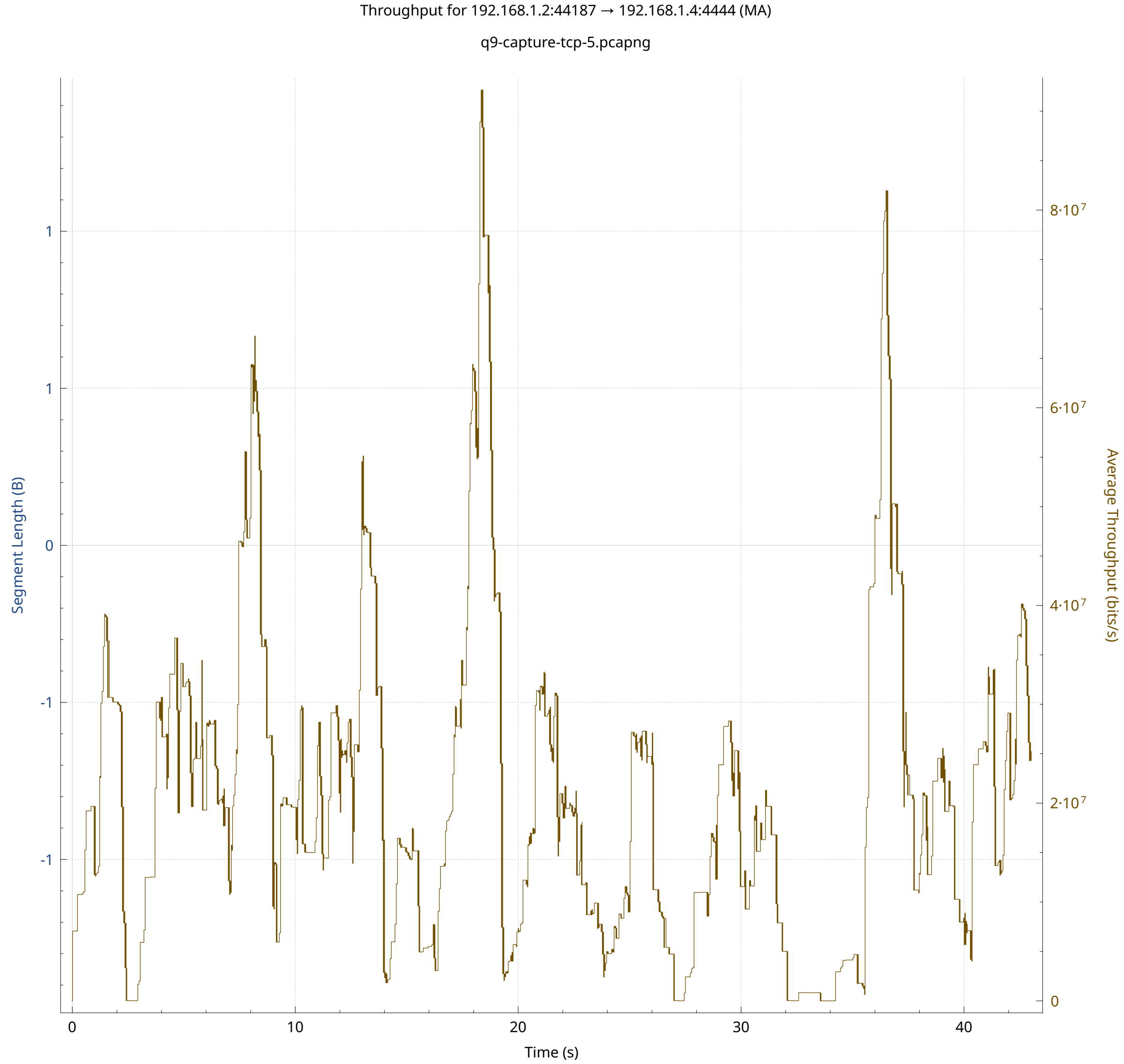


Figure 22: A plot of the 1-second MA of throughput against time (5% loss).

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.

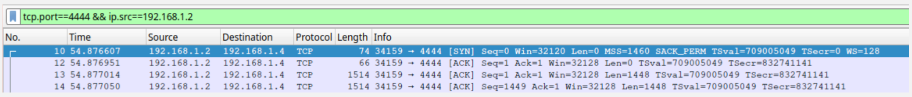
```
import argparse
import csv

def process_csv(file_path): try: with open(file_path, 'r')
as csv_file: reader = csv.reader(csv_file, delimiter=',',
```

```
quotechar='"') data = [(float(row[1]), int(row[5])) for row
in [row for row in reader][1:]]

packet_count = len(data) total_size = sum((length for _,
length in data)) start_time = min((time for time, _ in
data)) end_time = max((time for time, _ in data)) diff_time
= end_time - start_time average_throughput =
total_size/diff_time file_size = 100*1024*1024
average_effective_throughput = file_size / diff_time

print( f"""General Info -> Packet Count: {packet_count}
Total Size (in Bytes): {total_size} Start Time (in
Seconds): {start_time} End Time (in Seconds): {end_time}
Difference (in Seconds): {diff_time} Average Throughput (in
Bytes/Second): {total_size}/{diff_time} =
{average_throughput}

Conversions (Throughput) -> Average Throughput (in
MegaBits/Second): {(average_throughput * 8) / (1000 ** 2)}
Average Throughput (in MegaBytes/Second):
{average_throughput / (1000 ** 2)}

Effective Throughput (For 100MibiByte File) -> Average
Effective Throughput (in Bytes/Second):
{file_size}/{diff_time} = {average_effective_throughput}

Conversions (Effective Throughput)-> Average Effective
Throughput (in MegaBits/Second):
{(average_effective_throughput * 8) / (1000 ** 2)} Average
Effective Throughput (in MegaBytes/Second):
{average_effective_throughput / (1000 ** 2)}""")

except FileNotFoundError: print(f"Error: File '{file_path}'
not found.") except Exception as e: print(f"An error
occurred: {e}")

def main(): parser =
argparse.ArgumentParser(description="Process a CSV file.")
parser.add_argument('file', metavar='FILE', help='Path to
the CSV file')

args = parser.parse_args() process_csv(args.file)

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.

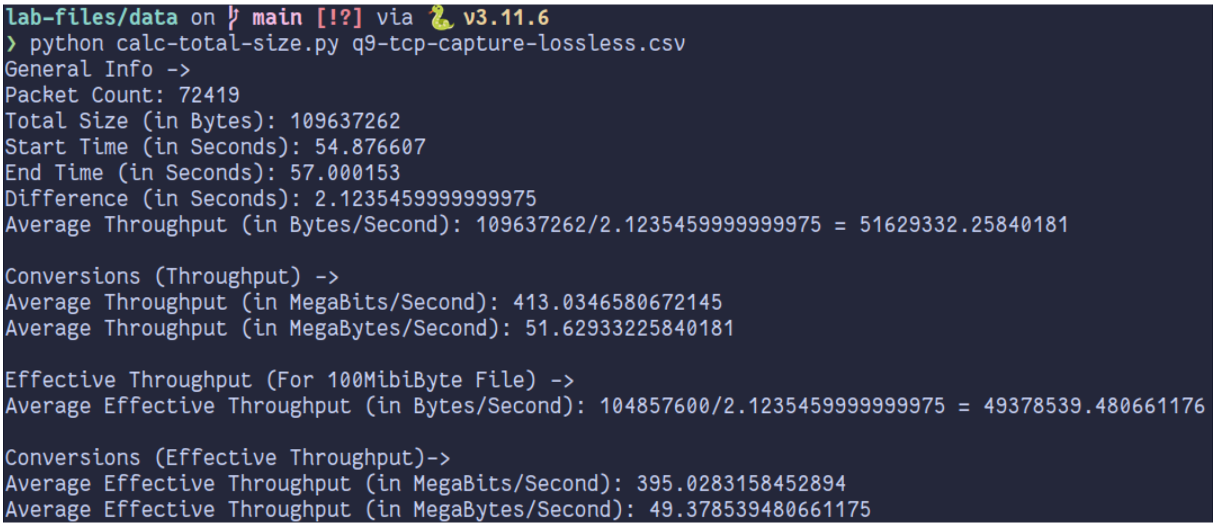
Note: The CSV files were generated from Wireshark with the following filter `tcp.port == 4444 && ip.src == 192.168.1.2` (see Figure ??) to ensure that only uni-directional throughput is considered.



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
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
13	54.877014	192.168.1.2	192.168.1.4	TCP	1514	34159 → 4444 [ACK] Seq=1 Ack=1 Win=32128 Len=1448 TSval=709005049 TSecr=832741141
14	54.877050	192.168.1.2	192.168.1.4	TCP	1514	34159 → 4444 [ACK] Seq=1449 Ack=1 Win=32128 Len=1448 TSval=709005049 TSecr=832741141

Figure 23: A picture of the capture filter active for extracting the appropriate packets.

This is because as mentioned on the Lab Forum, twisted pair cables are full duplex. This means that traffic can flow in both directions simultaneously without impacting the throughput of each direction.



```
lab-files/data on  main [!?] via 🐍 v3.11.6
❯ python calc-total-size.py q9-tcp-capture-lossless.csv
General Info ->
Packet Count: 72419
Total Size (in Bytes): 109637262
Start Time (in Seconds): 54.876607
End Time (in Seconds): 57.000153
Difference (in Seconds): 2.1235459999999975
Average Throughput (in Bytes/Second): 109637262/2.1235459999999975 = 51629332.25840181

Conversions (Throughput) ->
Average Throughput (in MegaBits/Second): 413.0346580672145
Average Throughput (in MegaBytes/Second): 51.62933225840181

Effective Throughput (For 100MibiByte File) ->
Average Effective Throughput (in Bytes/Second): 104857600/2.1235459999999975 = 49378539.480661176

Conversions (Effective Throughput)->
Average Effective Throughput (in MegaBits/Second): 395.0283158452894
Average Effective Throughput (in MegaBytes/Second): 49.378539480661175
```

Figure 24: The statistics generated by the Python script for lossless transfer.

```
lab-files/data on  main [!?] via 🐍 v3.11.6
❯ python calc-total-size.py q9-tcp-capture-loss1.csv
General Info ->
Packet Count: 73873
Total Size (in Bytes): 111838618
Start Time (in Seconds): 4.025614
End Time (in Seconds): 7.075828
Difference (in Seconds): 3.0502139999999995
Average Throughput (in Bytes/Second): 111838618/3.0502139999999995 = 36665826.72560024

Conversions (Throughput) ->
Average Throughput (in MegaBits/Second): 293.3266138048019
Average Throughput (in MegaBytes/Second): 36.66582672560024

Effective Throughput (For 100MibiByte File) ->
Average Effective Throughput (in Bytes/Second): 104857600/3.0502139999999995 = 34377128.94898523

Conversions (Effective Throughput)->
Average Effective Throughput (in MegaBits/Second): 275.0170315918818
Average Effective Throughput (in MegaBytes/Second): 34.377128948985224
```

Figure 25: The statistics generated by the Python script for 1% loss transfer.

```
lab-files/data on  main [!?] via 🐍 v3.11.6
❯ python calc-total-size.py q9-tcp-capture-loss3.csv
General Info ->
Packet Count: 75539
Total Size (in Bytes): 114362390
Start Time (in Seconds): 19.659794
End Time (in Seconds): 30.606881
Difference (in Seconds): 10.947087
Average Throughput (in Bytes/Second): 114362390/10.947087 = 10446833.025077812

Conversions (Throughput) ->
Average Throughput (in MegaBits/Second): 83.5746642006225
Average Throughput (in MegaBytes/Second): 10.446833025077812

Effective Throughput (For 100MibiByte File) ->
Average Effective Throughput (in Bytes/Second): 104857600/10.947087 = 9578584.695636382

Conversions (Effective Throughput)->
Average Effective Throughput (in MegaBits/Second): 76.62867756509105
Average Effective Throughput (in MegaBytes/Second): 9.578584695636382
```

Figure 26: The statistics generated by the Python script for 3% loss transfer.

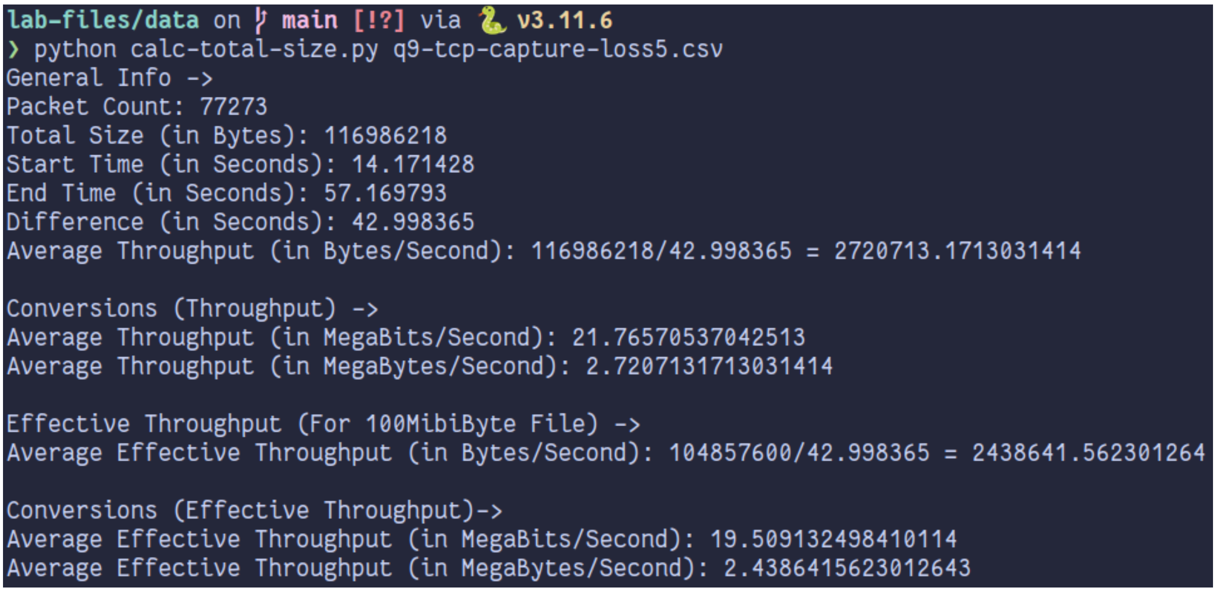


Figure 27: The statistics generated by the Python script for 5% loss transfer.

> **Note:** The Python script `calc-total-size.py` 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	51.63	49.38
1% Loss	36.67	34.38
3% Loss	10.45	9.58
5% Loss	2.72	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* 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 28: 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 29: Screenshot of the last three TCP packets from the lossless capture.

5. From Figures 28 & 29 the range of the packets is 10 to 108331.
6. The packets that form part of the teardown are those in Figure 29.

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

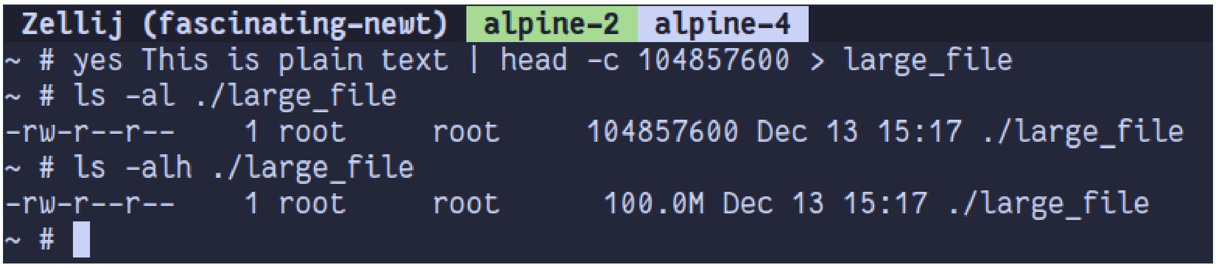


Figure 30: Large file generation on alpine-2.

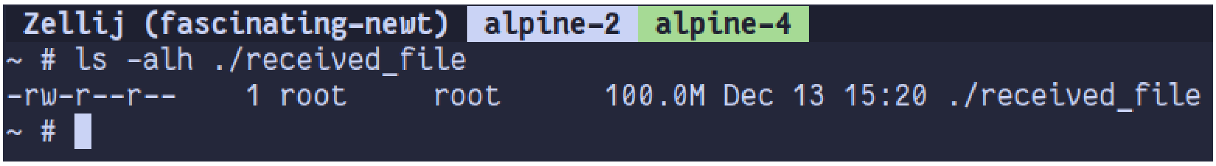


Figure 31: Listing received large file on alpine-4.

As can be seen from Figures 30 & 31 the file size remained the same. This is the case 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

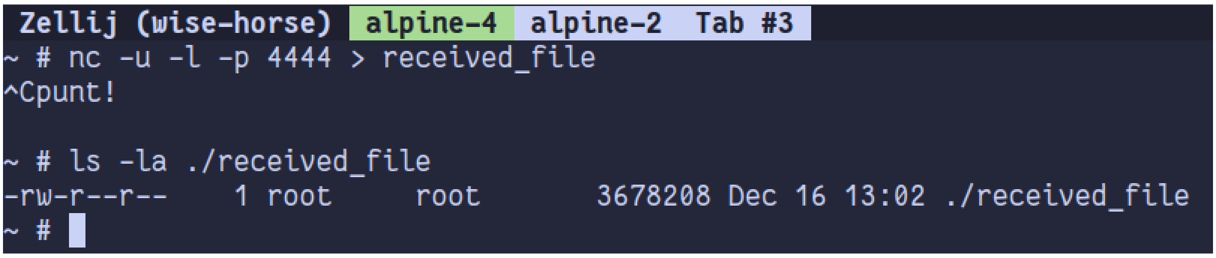


Figure 32: Listing received large file on alpine-4 for Question 12.1.

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.

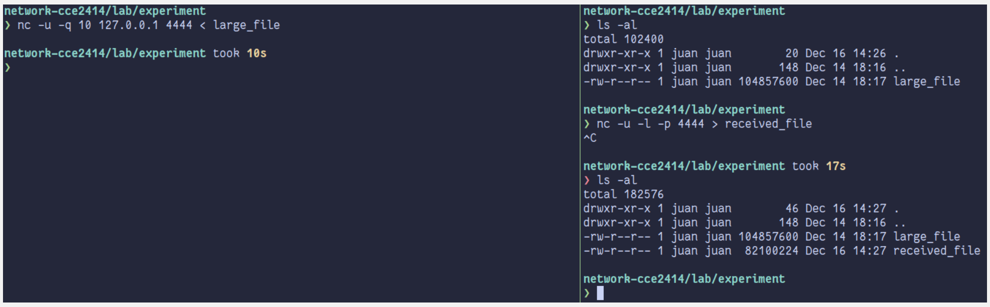


Figure 33: Using Netcat in UDP mode for local file transfer using the Loopback device.

As can be seen from Figure 33 the size of `received_file` is 82100224 bytes. This is $100 \times \frac{82100224}{104857600} \approx 78.3\%$ of the original file size; a better and 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 Figure 33 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.

```
import argparse
import csv

def process_csv(file_path): try: with open(file_path, 'r')
as csv_file: reader = csv.reader(csv_file, delimiter=',',
quotechar='"') data = [(float(row[1]), int(row[5])) for row
in [row for row in reader][1:]]

packet_count = len(data) total_size = 8192 * packet_count
start_time = min((time for time, _ in data)) end_time =
max((time for time, _ in data)) diff_time = end_time -
start_time

print( f"""General Info -> Packet Count: {packet_count}
Total Size (in Bytes): {total_size} Start Time (in
Seconds): {start_time} End Time (in Seconds): {end_time}
```

```
Difference (in Seconds): {diff_time}

Conversions -> Total Size (in MibiBytes): {(total_size) /
(1024 ** 2)}""")

except FileNotFoundError: print(f"Error: File '{file_path}'
not found.") except Exception as e: print(f"An error
occurred: {e}")

def main(): parser =
argparse.ArgumentParser(description="Process a CSV file.")
parser.add_argument('file', metavar='FILE', help='Path to
the CSV file')

args = parser.parse_args() process_csv(args.file)

if __name__ == "__main__": main()
```

Note: The Python script `udp-total-size.py` and CSV file used are present in the `data` directory.

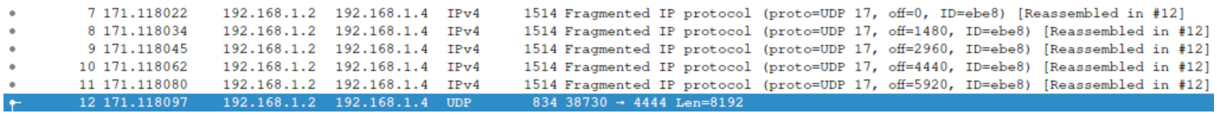
```
tex-files/data on  main [!?] via 🐍 v3.11.6
› python udp-total-size.py q12-udp-capture-lossless.csv
General Info ->
Packet Count: 847
Total Size (in Bytes): 6938624
Start Time (in Seconds): 171.118097
End Time (in Seconds): 171.245407
Difference (in Seconds): 0.12730999999999426

Conversions ->
Total Size (in MibiBytes): 6.6171875
```

Figure 34: Information about the UDP Wireshark capture (lossless).

Again, these calculations where 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 34 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 reorder and recombined to reconstruct the UDP packet. This is also indicated to us by Wireshark, see Figure 35.



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 35: A subsection of the UPD 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 answers to 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 give any guarantees about the reliability of data transfer. In fact, it is possible that none of the data sent over UDP, actually reaches its intended destination. Additionally, UDP does not 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 protocol due to reduced protocol overhead. Being leaner implies a greater effective throughput *i.e.* it can transfer data at faster rate.

UDP is leaner because of the fact that it does not need to establish a connection and more importantly it does not need to maintain said connection. This makes UDP more suited for applications with continuous streams of data generation which are tolerant of some missing data and require low-latency. 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.

Finally, to not leave TCP out, it unsurprisingly has a many more applications since it is the backbone of just about any form of reliable transfer: file transfers, emails, HTTP (and by proxy web pages) and more.