

Computer-Networking Course Final Project

Due to size of the file, we will submit the project through our git repository which can be found in the following link:

<https://github.com/yuvili/FinalProject>

Introduction

The goal of this project is to develop a system that combines three essential servers: a DHCP server, a DNS server, and an HTTP server.

The DHCP server provides the option for clients to claim an IP address and release it when no longer needed.

The DNS server enables clients to send DNS queries and receive responses.

Lastly, the HTTP server is designed to redirect clients to a specific image through a URL.

The DHCP server assigns IP addresses dynamically, ensuring efficient use of available resources, while the DNS server simplifies the process of locating resources by mapping human-readable domain names to IP addresses. The HTTP server redirects clients to the desired image, enabling easy access to digital content.

Overall, this project aims to provide a comprehensive networking solution that simplifies the process of connecting to the network and accessing resources.

The main process of the program is the “main.py” file located in src/frontend folder.

By default, the user will be initialized with the IP address “0.0.0.0” and their Mac address will be extracted from their device.

In order to claim an IP address, the user uses the DHCP option tab, where an IP address is generated.

In case the user chooses to stay with the default address, the DNS option tab will be disabled, since an IP address in requires to make a DNS query.

At any given time, the user can open “Client Info” tab to see their computer’s information.

Dynamic Host Configuration Protocol (DHCP)

DHCP (Dynamic Host Configuration Protocol) is a network protocol used to dynamically assign IP addresses and other configuration parameters to network devices.

The following image shows a DHCP message format, which is based on the BOOTP message format although DHCP uses some of the fields in significantly different ways. The numbers in parentheses indicate the size of each field in bytes.

0	7	15	23	31			
op (1)		htype (1)		hlen (1)		hops (1)	
xid (4)							
secs (2)				flags (2)			
ciaddr (4)							
yiaddr (4)							
siaddr (4)							
giaddr (4)							
chaddr (16)							
sname (64)							
file (128)							
options (variable)							

- **op:** Message type (1 is a REQUEST, 2 for REPLY).
- **htype, hlen:** Hardware address type and length of a DHCP Client.
- **hops:** Number of relay agents a request message traveled.
- **xid:** Transaction ID, a random number chosen by the client for each transaction.
- **secs:** Filled in by the client, the number of seconds elapsed since the client began address acquisition or renewal process.
- **flags:** If this flag is set to 0, the DHCP server sent a reply back by unicast, if this flag is set to 1, the DHCP server sent a reply back by broadcast. The remaining bits of the flags field are reserved for future use.
- **ciaddr:** Client IP address.
- **yiaddr:** 'your' (client) IP address, assigned by the server.
- **siaddr:** Server IP address, from which the client obtained configuration parameters.
- **giaddr:** IP address of the first relay agent a request message traveled.
- **chaddr:** Client Mac address.
- **sname:** Server host name.
- **file:** Bootfile name and path information, defined by the server to the client.
- **options:** Optional parameters field that is variable in length, which includes the message type, lease, domain name server IP address, etc..

DHCP server

This code implementing the Dynamic Host Configuration Protocol (DHCP), which is used to automatically assign IP addresses and other network configuration parameters to network devices.

The code defines three functions: `offer()`, `ack()` and `nak()`, which respectively handle the DHCP Offer, DHCP ACK and DHCP NAK messages.

- **`offer()`**: takes in a `packet` parameter, which is a DHCP Discovery message sent by a DHCP client. It begins by unpacking the message using the `struct.unpack()` method to retrieve the different fields of the message. After unpacking the message, the function generates an IP address to offer to the client by randomly choosing an available address from a list of `available_address`. It then sets the values for the different fields of the DHCP Offer message, using the `struct.pack()` method to pack them into a binary string. Next, the function creates a UDP socket using the `socket()` method, binds it to the DHCP server port (67), and sets it to broadcast mode. It then constructs the full DHCP Offer packet by concatenating the offer header and the DHCP options. Finally, the function sends the DHCP Offer packet to the broadcast address on UDP port 68 (the DHCP client port) using the `sendto()` method of the socket object. It then closes the socket using the `close()` method.
- **`ack()`**: takes in a `request_packet` parameter, which is a DHCP Request message sent by a DHCP client. It begins by checking if the request was sent to this DHCP server by comparing the server ID field of the message to the IP address of this DHCP server. If the request was sent to this server, the function unpacks the message using the `struct.unpack()` method to retrieve the different fields of the message. It then checks if the requested IP address is available by checking if it is in the list of available addresses. If the requested address is not available, the function sends a DHCP NAK message using the `nak()` function and returns. If the requested address is available, the function sets the values for the different fields of the DHCP ACK message, using the `struct.pack()` method to pack them into a binary string. It then constructs the full DHCP ACK packet by concatenating the ACK header and the DHCP options. Finally, the function sends the DHCP ACK packet to the broadcast address on UDP port 68 (the DHCP client port) using the `sendto()` method of a new socket object. It then closes the socket using the `close()` method.

DHCP client

This code defines a class ClientDHCP for a DHCP client to communicate with a DHCP server to obtain an IP address lease. The class has several instance variables to hold the client's network configuration, including the MAC address, IP address, DNS server address, subnet mask, router, and lease information.

The client is initially initialized with the default IP "0.0.0.0", and their Mac address is extracted.

The class has five methods:

- **discover():** Sends a DHCP discover packet to the network's broadcast address to find a DHCP server that can provide an IP address lease. The method builds the DHCP discover packet and sends it to the broadcast address on UDP port 68 (DHCP client port) using a UDP socket. After sending the packet, the client waits for a DHCP offer packet from the server. When the client receives an offer, it calls the `set_offer()` method to extract the offered IP address and other lease details.
- **set_offer(offer_packet):** Extracts the offered IP address and other lease details from the DHCP offer packet sent by the server. The method unpacks the packet's header and extracts the server's IP address and offered IP address.
- **request():** Sends a DHCP request packet to the server to request the offered IP address lease. The method builds the DHCP request packet and sends it to the server using the UDP socket. After sending the packet, the client waits for a DHCP acknowledgment packet from the server. If the server acknowledges the lease, the client sets the lease information in its instance variables.
- **decline():** Sends a DHCP decline packet to the server to decline the offered IP address lease.
- **release():** Sends a DHCP release packet to the server to release the claimed IP address. In this case, the client will remain "0.0.0.0".

Wireshark Recordings

Since the packets were built using `socket()`, the actual IP shown in the Wireshark recordings is our actual IP address, since `socket()` sends packets in the app layer, and the actual IP couldn't be changed.

In the recording after claiming an IP address (as a result of discover-offer-request-ack process), we released it with DHCP Release. We then made another DHCP Discover, got a DHCP offer but we sent a DHCP Decline message.

We will examine the first discover-offer-request-ack and release process, and the decline message.

Discover – the client sent a DHCP Discover broadcast packet, in order to get an offer from a server.

- message type 1 indicated a DHCP request.
- Transaction ID was generated by the client.
- Client IP address (ciaddr), “your” client IP address (yiaddr), siaddr, giaddr where all initialized to “0.0.0.0”, since client lacks that information and waits to receive it from the server.
- Options include the DHCP message type which sets to 1 from a DHCP Discover message.

No.	Time	Source	Destination	Protocol	Length	Info
233	10.168575	192.168.1.181	255.255.255.255	DHCP	299	DHCP Discover - Transaction ID 0x45fde0c8

> Frame 233: 299 bytes on wire (2392 bits), 299 bytes captured (2392 bits) on interface	0000	ff ff ff ff ff ff b0 be 83 1e 0a 30 00 00 45 000.0.0.E			
> Ethernet II, Src: Apple_1e:0a:30 (b0:be:83:1e:0a:30), Dst: Broadcast (ff:ff:ff:ff:ff:ff)	0010	01 10 5e e7 00 00 40 11 58 8c c0 a8 01 b5 ff ff	...A...0: X.....			
> Internet Protocol Version 4, Src: 192.168.1.181, Dst: 255.255.255.255	0020	ff ff 00 44 00 43 01 00 cb 80 01 01 06 00 45 fd	...D.C.....E			
> User Datagram Protocol, Src Port: 68, Dst Port: 67	0030	e0 c8 00 00 00 00 00 00 00 00 00 00 00 00 000.....			
> Dynamic Host Configuration Protocol (Discover)	0040	00 00 00 00 00 00 b0 be 83 1e 0a 30 00 00 000.....			
Message type: Boot Request (1)	0050	00 00 00 00 00 00 00 00 00 00 00 00 00 00 000.....			
Hardware type: Ethernet (0x01)	0060	00 00 00 00 00 00 00 00 00 00 00 00 00 00 000.....			
Hardware address length: 6	0070	00 00 00 00 00 00 00 00 00 00 00 00 00 00 000.....			
Hops: 0	0080	00 00 00 00 00 00 00 00 00 00 00 00 00 00 000.....			
Transaction ID: 0x45fde0c8	0090	00 00 00 00 00 00 00 00 00 00 00 00 00 00 000.....			
Seconds elapsed: 0	00a0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 000.....			
Bootp flags: 0x0000 (Unicast)	00b0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 000.....			
Client IP address: 0.0.0.0	00c0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 000.....			
Your (client) IP address: 0.0.0.0	00d0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 000.....			
Next server IP address: 0.0.0.0	00e0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 000.....			
Relay agent IP address: 0.0.0.0	00f0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 000.....			
Client MAC address: Apple_1e:0a:30 (b0:be:83:1e:0a:30)	0100	00 00 00 00 00 00 63 82 53 63 35 01 01 3d 06 b0c.Sc5...=			
Client hardware address padding: 00000000000000000000	0110	00 00 00 00 00 00 63 82 53 63 35 01 01 3d 06 b0c.Sc5...=			
Server host name not given	0120	be 83 1e 0a 30 37 03 03 01 06 ff07.....			
Boot file name not given						
Magic cookie: DHCP						
> Option: (53) DHCP Message Type (Discover)						
Length: 1						
DHCP: Discover (1)						
> Option: (61) Client identifier						
Length: 6						
> Option: (55) Parameter Request List						
Length: 3						
Parameter Request List Item: (3) Router						
Parameter Request List Item: (1) Subnet Mask						
Parameter Request List Item: (6) Domain Name Server						
> Option: (255) End						
Option End: 255						

Offer – the server received the discover message, copied most of the fields from the client packet, except for the following:

- message type 2 indicated a DHCP replay.
- “your” client IP address (yiaddr) changed by the server to be the offered address, in the example below is set to be “10.0.0.209”.
- Siaddr change by the server to be the server’s own IP address (“10.0.0.1”).
- giaddr change by the server to be the router’s IP address (“10.0.0.1”).
- Options include the DHCP message type which sets to 2 from a DHCP Offer message, added with the server’s IP, the lease of the offered IP, the renewal time value which

The server add the Subnet mask, broadcast address, router address and DNS server address in addition.

Request – the client sent a DHCP Request packet, in order to get an offer from a server.

- | No. | Time | Source | Destination | Protocol | Length | Info |
|-----|-----------|---------------|-----------------|----------|--------|--|
| 267 | 11.313888 | 192.168.1.181 | 255.255.255.255 | DHCP | 298 | DHCP Request - Transaction ID 0x45fde0c8 |


```

> Frame 267: 298 bytes on wire (2384 bits), 298 bytes captured (2384 bits) on interface eth0
> Ethernet II, Src: Apple1e:8a:30 (b0:be:83:1e:8a:30), Dst: Broadcast (ff:ff:ff:ff:ff:ff)
> Internet Protocol Version 4, Src: 192.168.1.181, Dst: 255.255.255.255
> User Datagram Protocol, Src Port: 68, Dst Port: 67
Dynamic Host Configuration Protocol (Request)
  Message type: Boot Request (1)
  Hardware type: Ethernet (0x01)
  Hardware address length: 6
  Hops: 0
  Transaction ID: 0x45fde0c8
  Seconds elapsed: 0
  Bootp flags: 0x0000 (Unicast)
  Client IP address: 0.0.0.0
  Your (client) IP address: 0.0.0.0
  Next server IP address: 0.0.0.0
  Relay agent IP address: 0.0.0.0
  Client MAC address: Apple1e:8a:30 (b0:be:83:1e:8a:30)
  Client hardware address padding: 00000000000000000000
  Server host name not given
  Boot file name not given
  Magic cookie: DHCP
  Option: (53) DHCP Message Type (Request)
    Length: 1
    DHCP: Request (3)
  Option: (54) DHCP Server Identifier (10.0.0.1)
    Length: 4
    DHCP Server Identifier: 10.0.0.1
  Option: (50) Requested IP Address (10.0.0.209)
    Length: 4
    Requested IP Address: 10.0.0.209
  Option: (255) End
    Option End: 255
  
```


Offset	Hex	ASCII	
0000	ff ff ff ff ff ff b0 be	83 1e 0a 30 00 00 45 000..E..
0010	01 1c 15 8a 00 00 00 11	a1 ea c0 a8 01 b5 ff ff@.....
0020	ff ff 00 44 00 43 01 08	04 c3 01 01 06 00 45 fd	...D.C.....E..
0030	e0 c8 00 00 00 00 00 00	00 00 00 00 00 00 00 000.....
0040	00 00 00 00 00 00 b0 be	83 1e 0a 30 00 00 00 000.....
0050	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
0060	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
0070	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
0080	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
0090	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
00a0	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
00b0	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
00c0	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
00d0	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
00e0	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
00f0	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
0100	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
0110	00 00 00 00 00 63 82	53 63 35 01 03 36 04 0ac..Sc5..6..
0120	00 00 01 32 04 0a 00 00	d1 ff	...2.....

ACK- the final packet in this transaction. This is the server’s approval of the requested address. To this packet the server added the same information listed in the Offer message, with a change of the message type to 5 for DHCP ACK package.

```

268 11.314443 192.168.1.181 255.255.255.255 DHCP 334 DHCP ACK - Transaction ID 0x45fde0c8
> Frame 268: 334 bytes on wire (2672 bits), 334 bytes captured (2672 bits) on interface
> Ethernet II, Src: Apple_1e:0a:30 (b0:be:83:1e:0a:30), Dst: Broadcast (ff:ff:ff:ff:ff:ff)
> Internet Protocol Version 4, Src: 192.168.1.181, Dst: 255.255.255.255
> User Datagram Protocol, Src Port: 67, Dst Port: 68
> Dynamic Host Configuration Protocol (ACK)
  Message type: Boot Reply (2)
  Hardware type: Ethernet (0x01)
  Hardware address length: 6
  Hops: 0
  Transaction ID: 0x45fde0c8
  Seconds elapsed: 0
  > Bootp flags: 0x0000 (Unicast)
  Client IP address: 0.0.0.0
  Your (client) IP address: 10.0.0.209
  Next server IP address: 10.0.0.1
  Relay agent IP address: 10.0.0.1
  Client MAC address: Apple_1e:0a:30 (b0:be:83:1e:0a:30)
  Client hardware address padding: 00000000000000000000
  Server host name not given
  Boot file name not given
  Magic cookie: DHCP
  > Option: (53) DHCP Message Type (ACK)
    Length: 1
    DHCP: ACK (5)
  > Option: (54) DHCP Server Identifier (10.0.0.1)
    Length: 4
    DHCP Server Identifier: 10.0.0.1
  > Option: (51) IP Address Lease Time
    Length: 4
    IP Address Lease Time: (86400s) 1 day
  > Option: (58) Renewal Time Value
    Length: 4
    Renewal Time Value: (43200s) 12 hours
  > Option: (59) Rebinding Time Value

```

Release – the client sent to the broadcast address the release message to informed of the intent to release the IP address.

- message type 1 indicated a DHCP request.
- Transaction ID was generated by the client.
- Client IP address is set to the previous IP address given by the server (“10.0.0.209”).
- Options include the DHCP message type which sets to 1 from a DHCP Discover message.

```

303 12.798824 192.168.1.181 255.255.255.255 DHCP 286 DHCP Release - Transaction ID 0x20af653f
> Frame 303: 286 bytes on wire (2288 bits), 286 bytes captured (2288 bits) on interface
> Ethernet II, Src: Apple_1e:0a:30 (b0:be:83:1e:0a:30), Dst: Broadcast (ff:ff:ff:ff:ff:ff)
> Internet Protocol Version 4, Src: 192.168.1.181, Dst: 255.255.255.255
> User Datagram Protocol, Src Port: 67, Dst Port: 68
> Dynamic Host Configuration Protocol (Release)
  Message type: Boot Request (1)
  Hardware type: Ethernet (0x01)
  Hardware address length: 6
  Hops: 0
  Transaction ID: 0x20af653f
  Seconds elapsed: 0
  > Bootp flags: 0x0000 (Unicast)
  Client IP address: 10.0.0.209
  Your (client) IP address: 10.0.0.209
  Next server IP address: 10.0.0.1
  Relay agent IP address: 10.0.0.1
  Client MAC address: Apple_1e:0a:30 (b0:be:83:1e:0a:30)
  Client hardware address padding: 00000000000000000000
  Server host name not given
  Boot file name not given
  Magic cookie: DHCP
  > Option: (53) DHCP Message Type (Release)
    Length: 1
    DHCP: Release (7)
  > Option: (255) End
    Option End: 255

```

As shown in the recording the real DHCP also sent an offer to the client, but then when it detected that the client chose another server’s offer, the real DHCP server sent a NAK message with the message “wrong server id”, since the client’s request packet contained our server’s id.

```

390 16.217978      192.168.1.181      255.255.255.255      DHCP      334 DHCP Offer      - Transaction ID 0xc98e4e03
> Frame 390: 334 bytes on wire (2672 bits), 334 bytes captured (2672 bits) on interface0
> Ethernet II, Src: Apple1e:0a:30 (b0:be:83:1e:0a:30), Dst: Broadcast (ff:ff:ff:ff:ff:ff)
> Internet Protocol Version 4, Src: 192.168.1.181, Dst: 255.255.255.255
> User Datagram Protocol, Src Port: 67, Dst Port: 68
> Dynamic Host Configuration Protocol (Offer)
  Message type: Boot Reply (2)
  Hardware type: Ethernet (0x01)
  Hardware address length: 6
  Hops: 0
  Transaction ID: 0xc98e4e03
  Seconds elapsed: 0
  > Bootp flags: 0x0000 (Unicast)
  Client IP address: 0.0.0.0
  Your (client) IP address: 10.0.0.50
  Next server IP address: 10.0.0.1
  Relay agent IP address: 10.0.0.1
  Client MAC address: Apple1e:0a:30 (b0:be:83:1e:0a:30)
  Client hardware address padding: 00000000000000000000
  Server host name not given
  Boot file name not given
  Magic cookie: DHCP
  > Option: (53) DHCP Message Type (Offer)
    Length: 1
    DHCP: Offer (2)
  > Option: (54) DHCP Server Identifier (10.0.0.1)
    Length: 4
    DHCP Server Identifier: 10.0.0.1
  > Option: (51) IP Address Lease Time
    Length: 4
    IP Address Lease Times: (86400s) 1 day
  > Option: (58) Renewal Time Value
    Length: 4
    Renewal Time Value: (43200s) 12 hours
  > Option: (59) Rebinding Time Value
    Length: 4

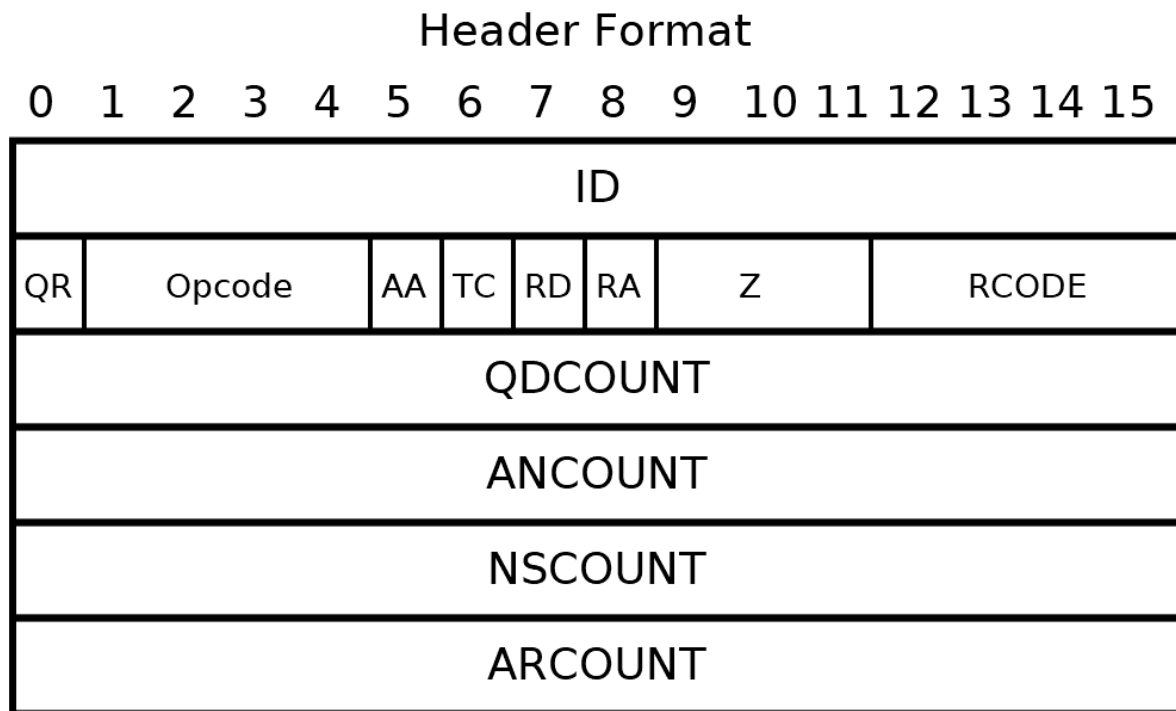
```

No.	Time	Source	Destination	Protocol	Length	Info
412	17.179020	192.168.1.181	255.255.255.255	DHCP	292	DHCP Decline - Transaction ID 0xc98e4e03
>	Frame 412:	292 bytes on wire (2336 bits), 292 bytes captured (2336 bits) on interface		0000 ff ff ff ff ff ff b0 be 83 1e 0a 30 08 00 45 000-E-		
>	Ethernet II, Src:	Apple_1e:0a:30 (b0:be:83:1e:0a:30), Dst: Broadcast (ff:ff:ff:ff:ff:ff)		0010 01 16 27 18 00 00 40 11 00 62 c0 a8 01 b5 ff ff0-b.....		
>	User Datagram Protocol, Src Port:	68, Dst Port: 67		0020 ff ff 00 44 00 43 01 02 c8 3f 01 01 06 00 c9 8eD C-7.....		
>	Dynamic Host Configuration Protocol (Decline)			0030 4c 03 00 00 00 00 00 00 00 00 00 00 00 00 00N.....		
>	Message type: Boot Request (1)			0040 00 00 00 00 00 00 b0 be 83 1e 0a 30 00 00 00 000.....		
>	Hardware type: Ethernet (0x01)			0050 00 00 00 00 00 00 00 00 00 00 00 00 00 00 000.....		
>	Hardware address length: 6			0060 00 00 00 00 00 00 00 00 00 00 00 00 00 00 000.....		
>	Hops: 0			0070 00 00 00 00 00 00 00 00 00 00 00 00 00 00 000.....		
>	Transaction ID: 0xc98e4e03			0080 00 00 00 00 00 00 00 00 00 00 00 00 00 00 000.....		
>	Seconds elapsed: 0			0090 00 00 00 00 00 00 00 00 00 00 00 00 00 00 000.....		
>	Bootp flags: 0x0000 (Unicast)			00a0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 000.....		
>	Client IP address: 0.0.0.0			00b0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 000.....		
>	Your (client) IP address: 0.0.0.0			00c0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 000.....		
>	Next server IP address: 0.0.0.0			00d0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 000.....		
>	Relay agent IP address: 0.0.0.0			00e0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 000.....		
>	Client MAC address: Apple_1e:0a:30 (b0:be:83:1e:0a:30)			00f0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 000.....		
>	Client hardware address padding: 00000000000000000000			0100 00 00 00 00 00 00 00 00 00 00 00 00 00 00 000.....		
>	Server host name not given			0110 00 00 00 00 00 63 82 53 63 35 01 04 36 04 0ac-5C5-6.....		
>	Boot file name not given			0120 00 00 01 ff0.....		
>	Magic cookie: DHCP					
>	Options (53) DHCP Message Type (Decline)					
>	Length: 1					
>	DHCP: Decline (4)					
>	Option: (54) DHCP Server Identifier (10.0.0.1)					
>	Length: 4					
>	DHCP Server Identifier: 10.0.0.1					
>	Option: (255) End					
>	Option End: 255					

Domain Name Server (DNS)

DNS (Domain Name System) is a protocol used for translating human-readable domain names into their corresponding IP addresses, allowing clients to access servers and websites using easy-to-remember domain names instead of numerical IP addresses.

The following image shows a DNS message format:



- **ID (16 bit):** identifier assigned by the client. This identifier is copied to the corresponding reply.
- **QR (1 bit):** field that specifies whether this message is a query (0), or a response (1).
- **OPCODE (4 bit):** field that specifies kind of query in this message.
- **AA - Authoritative Answer (1 bit):** only relevant in responses, and specifies that the responding name server is an authority for the domain name.
- **TC - Truncation (1 bit):** specifies that this message was truncated.
- **RD - Recursion Desired (1 bit):** directs the name server to pursue the query recursively.
- **RA - Recursion Available (1 bit):** set or cleared in a response, and indicates whether recursive query support is available in the name server.
- **Z (3 bit):** Reserved for future use.
- **RCODE - Response code (4 bit):** set as part of responses indication of errors.
- **QDCOUNT (unsigned 16 bit):** specifying the number of entries in the question section.
- **ANCOUNT (unsigned 16 bit):** specifying the number of resource records in the answer section.
- **NSCOUNT (unsigned 16 bit):** specifying the number of name server resource records in the authority records section.

- **ARCOUNT (unsigned 16 bit):** specifying the number of resource records in the additional records section.

DNS Server

This code implements a DNS server that listens for DNS queries on port 53 and responds with the appropriate IP address for the requested domain name. The DNS server caches the resolved IP addresses for a period of time specified by the time-to-live (TTL) field in the DNS response.

The DNS server is implemented as a Python function **start_server()**, which creates a UDP socket bound to IP address "127.0.0.1" and port 53.

The **handle_dns_query()** function is called every time a DNS query is received on the socket. It parses the DNS query header to extract the query name and type, checks the DNS cache for a previously cached IP address for the query name, and if found, returns the cached IP address. If not found in the cache, the DNS server resolves the IP address using Python's `gethostbyname()` function, caches the resolved IP address for a TTL period of 1 day (86400 seconds) and returns the IP address in the DNS response.

The DNS response is constructed by packing the DNS response header and the DNS response data using the struct module. The DNS response header contains fields for the query ID, flags, question count, answer count, nameserver count, and additional record count. The DNS response data contains the query name, query type, answer type, TTL, and IP address.

DNS Client

This code defines a ClientDNS class that can send a DNS query to a DNS server and parse the server's response to obtain the IP address of a given hostname.

The main block of the code sets the hostname and query_type parameters, creates an instance of the ClientDNS class, and calls the send_dns_query method to obtain the IP address of the specified hostname. The obtained IP address is stored in the ip_result attribute of the ClientDNS object.

During the execution of the send_dns_query method, the code also prints some messages to the console to indicate when the query is sent and when the response is received.

The ClientDNS class has the following attributes:

- **client_port**: the client's port number, set to 57217.
- **server_port**: the DNS server's port number, set to 53.
- **ip_address**: a string that stores the IP address obtained after the DHCP process.
- **dns_server_add**: a string that stores the IP address of the DNS server obtained after the DHCP process.
- **subnet_mask**: a string that stores the IP address of the Subnet Mask obtained after the DHCP process.
- **router**: a string that stores the IP address of the Router obtained after the DHCP process.
- **ip_result**: a string that stores the IP address obtained from the DNS response.

The ClientDNS class has the following methods:

- **send_dns_query()**: this method takes two parameters: hostname and query_type. The hostname parameter is a string that contains the hostname to be resolved, and query_type is a string that specifies the type of DNS query (either 'A' or 'AAAA'). The method constructs a DNS query based on the provided parameters, sends it to the DNS server using a UDP socket, and waits for the server's response. Once the response is received, it calls the parse_dns_response method to extract the requested IP address.
- **parse_dns_response()**: this method takes three parameters: dns_response, query_data, and query_len. The dns_response parameter is the server's response to the DNS query, query_data is the client's original query, and query_len is the length of the original query. The method parses the DNS response and extracts the requested IP address, storing it in the ip_result attribute.

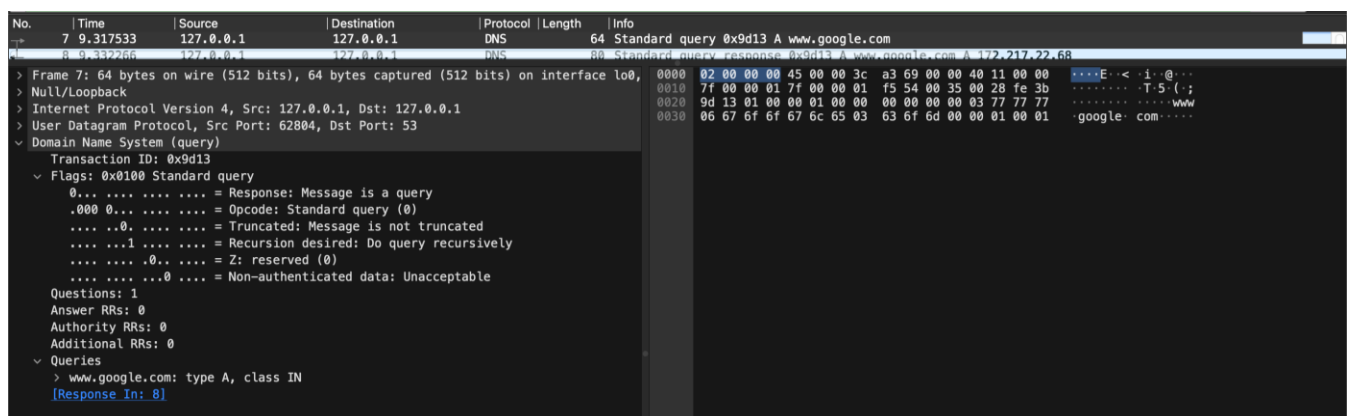
Wireshark recordings

In this recording we sent 3 DNS queries and got a DNS response for each of them respectfully. We will examine the first query that was sent.

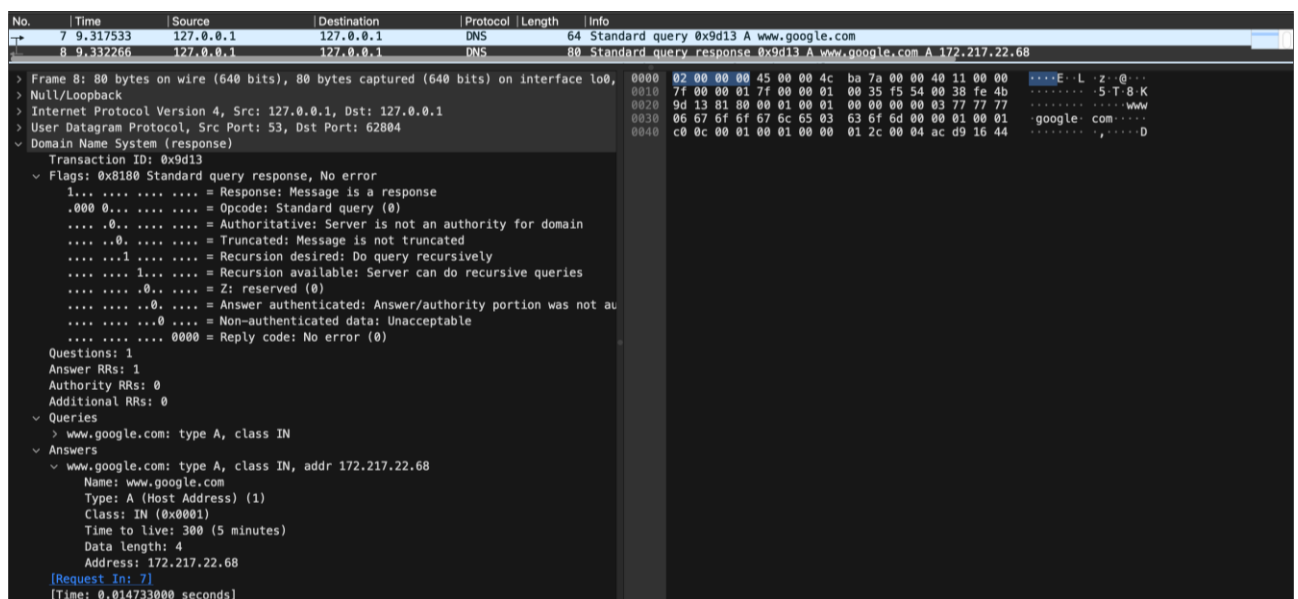
As shown in the picture below, the DNS query was sent from the client at port 62804, and sent to the DNS server with the address ("127.0.0.1", 53). The IP address was transferred to the client in the DHCP process.

In the "Flags" section of the packet we can see that it set for 0x0100 implying of a query message.

In the "Queries" section of the packet, it shows the required hostname and type 'A' for IPv4 type.



In the next line of the recording, we can see the server's response. The server copies the transaction ID and the Queries section but changes the "Flags" to be 0x8180 as an indication of a DNS response. Furthermore, the server added the answered IP in the "Answer" section of the packet.



HTTP application – using the transmission control protocol (TCP)

HTTP protocol:

HTTP is a protocol of the application layer used for surfing the Internet, it allows applications to access and use network resources on the Internet. The protocol is almost not used today, as it has been replaced by the HTTPS protocol (a more secure protocol). The HTTP protocol uses a communication method called "request-response", in which a client requests certain information from HTTP server by sending a certain request, and in response the HTTP server returns the relevant information. HTTP protocol uses TCP as its transport protocol, meaning that data is transmitted using TCP segments. TCP ensures that all data is transmitted reliably and in order, while HTTP specifies the format of the data being transmitted (such as HTML, CSS, or JavaScript). Together, TCP and HTTP form the backbone of the modern web, enabling fast and reliable transfer of data between web servers and clients.

About HTTP request:

The word GET indicates that it is an HTTP request of the GET type intended to fetch some item of information from the server on the Internet.

About HTTP response:

Each response consists of 3 main characteristics:

- **HTTP protocol version**
- **Status code** - describe the state of the answer, for example:
 - Status code 200 = OK
 - Status code 301 = the resource requested has moved Permanently(redirect)
 - Status code 302 = the resource requested has been temporarily moved(redirect)
 - Status code 400 = Bad Request
 - Status code 404 = Not Found
- **The type of content in the answer**

About TCP-

The Transmission Control Protocol (TCP) is a communication protocol that operates at the transport layer of the Internet Protocol (IP). It is responsible for ensuring reliable and ordered delivery of data between network applications running on different devices, which means that it ensures that all data is transmitted correctly and in order, even in the presence of packet loss, delay, or duplication.

The main ways that TCP achieves reliability is :

- **Segment numbers:** TCP establishing a virtual connection between two devices and exchanging data in the form of small packets called segments. Each segment is numbered and acknowledged by the receiving device to ensure that all data is delivered correctly and in order.
- **Flow control:** flow control regulates the rate at which data is transmitted between devices to prevent data overload or loss. TCP uses a sliding window algorithm to manage flow control, where the size of the window determines the number of segments that can be sent before waiting for an acknowledgment from the receiving device.
- **congestion control:** congestion control helps to prevent network congestion by controlling the rate at which data is transmitted between devices. TCP monitors network conditions, such as the number of packets in transit and the time it takes for packets to be acknowledged and adjusts the transmission rate accordingly to prevent congestion and packet loss. In the event of packet loss, TCP uses a mechanism called retransmission to ensure that all data is eventually transmitted correctly. When a device detects that a segment has been lost or corrupted, it retransmits the segment until it is acknowledged by the receiving device.

HTTP Server

This is a simple web server implemented in Python using socket programming. It handles with HTTP GET requests from client and respond with either an HTML file or an image file. The server contains the html file and send it to the client while receiving a HTML response, but this server doesn't have the png image file that needed for html file so while receiving png image file request the server redirect the client to another server (the image server) for downloading the image.

How it Works

The server uses the socket library to create a socket object that listens for incoming packets on a specific IP address and port number. When a client sends an HTTP GET request to the server, the server uses the threading library to create a new thread to handle the client request. The thread reads the request, processes it, and sends a response back to the client. If the client requests an HTML file, the server sends the HTML file back to the client. If the client requests an PNG image file, the server sends a redirection message to the client with the address of another server (the image server) that contains the image file. The client connects with the image server to download the file.

implementation

- This code defines a server that handles incoming client requests. The server listens on a specific IP address and port number for incoming TCP packets. When it receives a packet, it processes the data and sends back an appropriate response to the client.
- The server has a single thread that listens for incoming packets and handles them one at a time. When a client request arrives, the server creates a new thread to handle the request and then returns to listening for more packets.
- The code imports the socket module for socket programming and the threading module for creating new threads. It then defines several global variables, including the IP address and port number of the server, the IP address and port number of an image server, and the maximum amount of data (in bytes) that can be received at once.
- The **redirect()** function is called when a client request is received for an PNG image file. It sends a response to the client with a packet that includes the IP address and port number of the image server. This redirects the client to the image server, which then sends the requested image file to the client.
- The **client_handler()** function is called when a new thread is created to handle a client request. It receives data from the client, processes the request, and sends back an appropriate response. The function uses a while loop to keep listening for new packets until the socket is closed.
- The **start_server()** function is called to start the server. It creates a new socket using the **socket()** function and then sets the SO_REUSEADDR flag to prevent an "Address already in use" error. It then binds the server to a specific IP address and port number using the **bind()** function and starts listening for incoming packets using the **listen()** function. Finally, it enters an infinite loop to listen for new connections.

HTTP Image Server

This is a simple server written in Python that serves a single PNG image file to clients using the HTTP protocol.

How it Works

The server is implemented using a loop that listens for incoming connections and spawns a new thread to handle each client request. The `client_handler` function handles the communication with the client, receiving the request and sending the response. The image file is read from server and sent to the client in smaller parts of 1024 each until the client receive the whole image.

implementation

- This is a Python script for an HTTP server that serves an image file to clients over TCP/IP using socket programming.
- The script starts by importing the required modules for socket programming and threading. It also imports an image file from the `src/backend/HTTP_Docs/HTTP_Servers` directory as bytes to be sent to the clients.
- Then, it defines a function called `client_handler` which handles incoming client requests. This function receives a client's socket and address as parameters and it executes in an infinite loop until the client socket is closed.
- When a client sends a request to the server, the `client_handler` function receives the data, checks the request type and sends a response. If the client requests the image file, it sends an HTTP 200 OK response with the image file data. If it is a bad request, it sends an HTTP 400 Bad Request response.
- After the `client_handler` function is defined, the script defines another function called `start_server` which creates and binds the server socket to a specific IP address and port number. Then, it listens for incoming client connections and creates a new thread to handle each client request.
- Finally, the script starts the server by calling the `start_server` function in the main block, and it listens for incoming client connections until the script is interrupted with a keyboard interrupt signal (e.g., Ctrl+C).
- Overall, this script implements a basic HTTP server that can serve a single image file to multiple clients concurrently.

HTTP Client

This Python script provides a simple client that connects to a server to download an HTML file and then redirect to another server to download an PNG image file.

How it works

When a client requests a html file from the server using HTTP, the client and server establish a TCP connection by performing a three-way handshake. Once the connection is established, the client sends an HTTP request to the server, and the server responds with an HTTP response containing the requested data. Then the client sends a PNG image file request, but the server don't have it so it redirect the client to another server (the image server) so now the client start the process again this time with the image server until he finally download both files successfully. The client uses socket programming to establish a connection with the server and send HTTP requests. Specifically, it sends two requests:

An HTTP GET request for the HTML file:

```
GET / HTTP/1.1
Host: 127:0:0:1
Accept: text/html,application/xhtml+xml
Connection: keep-alive
```

If the server responds with a status of HTTP/1.1 200 OK (text/html), the client extracts the HTML content from the response and saves it to a new file named new_html.html. It then opens this file in a web browser and sends a request for the image file.

An HTTP GET request for the image file:

```
GET /imgs/OurImage.png HTTP/1.1
Host: 127:0:0:1
Accept: image/webp,*/*
Connection: keep-alive
```

If the server responds with a status of HTTP/1.1 200 OK (PNG image), the client extracts the image content from the response and saves it to a new file named OurImage.png. If the server responds with a status of HTTP/1.1 301 Moved Permanently, it means that the image file is located on a different server. The client extracts the new server's IP address and port from the response and establishes a new connection with the new server to download the image file.

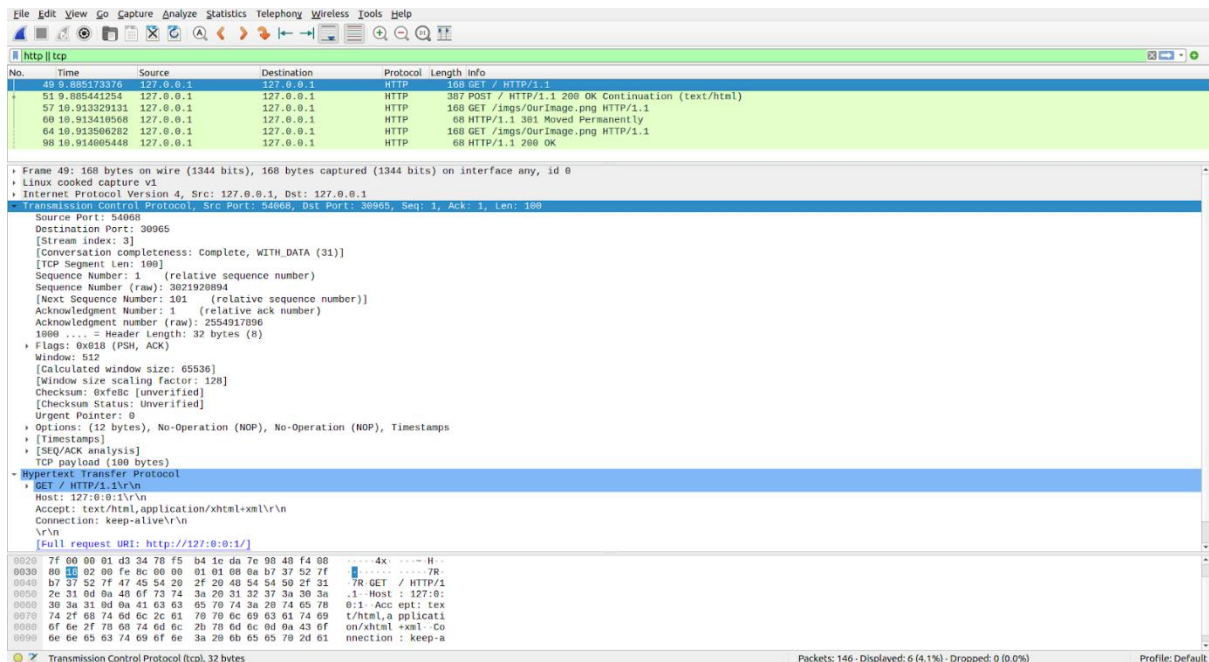
implementation

- This is a Python script that uses socket programming to send a GET request to a server and receive a response. The script sends a request for an HTML file and, upon receiving it, saves it to a new file named "new_html.html" and opens it in a new browser window. The script also sends a request for a PNG image file and saves it to a new file named "OurImage.png".
- The script starts by importing the necessary modules for socket programming, web browsing, and operating system interfacing. Three global variables are defined: the IP address of the server (127.0.0.1), the port number used by the server (30965), and the maximum amount of data that can be received at once (1024 bytes).
- Next, two functions are defined to create and send a GET request for an HTML file and a PNG image file. The `create_html_file_request()` function creates a GET request packet for the HTML file and returns it as a byte string. The `send_html_file_request()` function takes a client socket and server address as arguments, checks which server the client is currently connected to, and sends the HTML file request to the server. If the server accepts the request and sends the HTML file, the function receives the data from the socket, checks if the response status is "HTTP/1.1 200 OK (text/html)", saves the HTML data to a new file, opens the file in a new browser window, and sends a request for the PNG image file. If the server denies the request, the function sends the HTML file request again. If the client is currently connected to the image server, the function skips the HTML file request and sends a request for the PNG image file directly.
- The `create_image_request()` function creates a GET request packet for the PNG image file and returns it as a byte string. The `send_image_request()` function takes a client socket and server address as arguments, sends the PNG image file request to the server, receives the data from the socket, checks if the response status is "HTTP/1.1 200 OK (JPEG JFIF image)", and saves the image data to a new file.
- The main part of the script creates a client socket and connects to the server. It then calls the `send_html_file_request()` function to start the process of requesting and receiving files from the server. If an error occurs at any point, the script prints an error message to the console.

Wireshark recordings:

Note: These screenshots are taken from the Wireshark recordings "http_app_0.pcapng" and show the communication between the client with both servers during their connection when the packet loss percentage is zero.

Client html file request:



Explanation: In this recording we can see the HTTP file request message that the client sends to the server (HTTP server). It can be seen that the information is transferred using the TCP protocol, which enables reliability in the transfer of information between the client and the server. You can see that the server address is 127.0.0.1 and port number 30965 (as we defined it - with the last three digits being part of the end of one of our identity cards) and the client address is 127.0.0.1 and port 54068. You can see that this is a request for an html page by The word GET shows that the HTTP version is 1.1 and that the client expects to receive the HTML file back from the server.

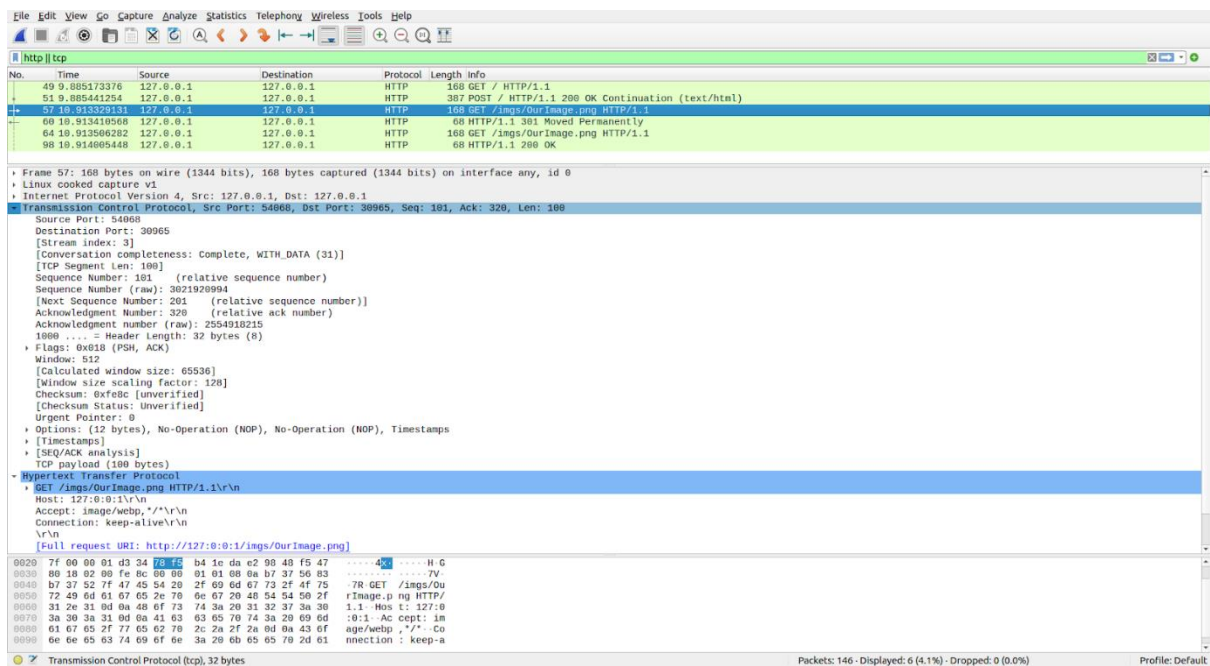
Server response - send the html file:

The screenshot displays the Wireshark interface with three main panes:

- Packets Pane (Left):** Shows a list of captured packets. Packet 26 is selected, which is an HTTP GET request from 127.0.0.1 to 127.0.0.1.
- Packet Details Pane (Middle):** Provides a hierarchical view of the selected packet's structure:
 - Ethernet II, Src: Intel (08:00:27:344A:5C), Dst: Intel (08:00:27:344A:5C)
 - Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
 - Hypertext Transfer Protocol, GET / HTTP/1.1
- Packet Bytes Pane (Right):** Displays the raw data of the selected packet in hexadecimal and ASCII format.

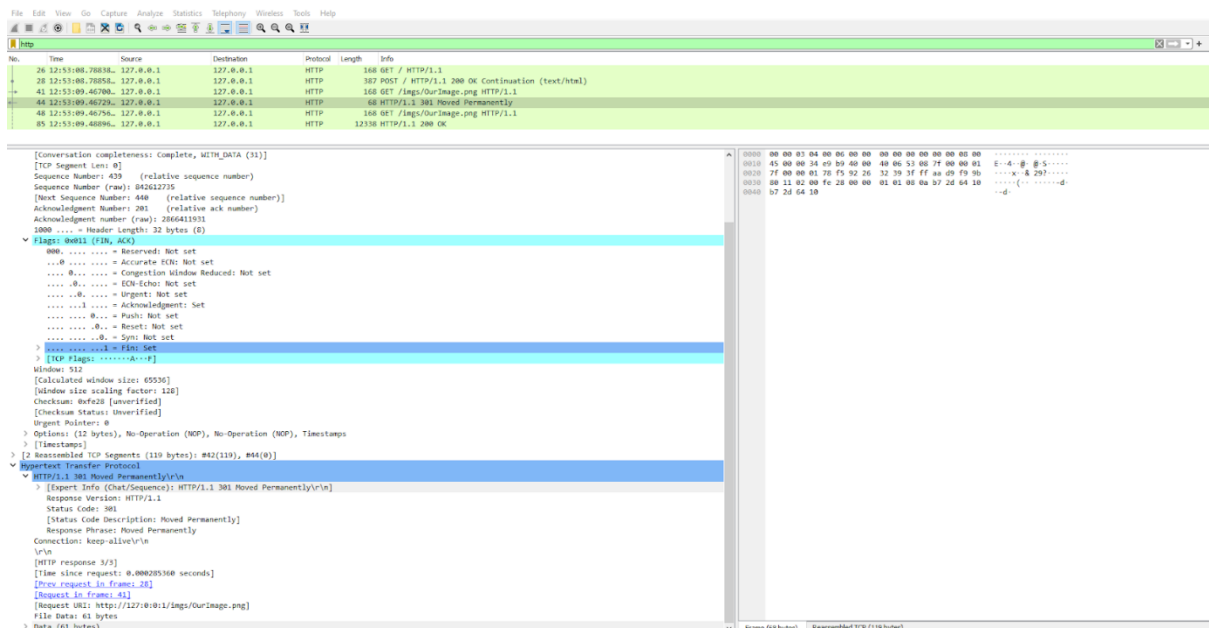
Explanation: In this recording we can see the response message that the server (HTTP server) sends to the client . It can be seen that the information is transferred using the TCP protocol, which enables reliability in the transfer of information between the client and the server. You can see that the server address is 127.0.0.1 and port number 30965 (as we defined it - with the last three digits being part of the end of one of our identity cards) and the client address is 127.0.0.1 and port 54068. You can see that this is a response by The “200 ok” status code and the HTTP version that is HTTP 1.1 and the type of the data that the server sends in this case text/html and we can see in the data section the html that was send.

Client request for the PNG image file:



Explanation: In this recording we can see the PNG image file request message that the client sends to the server (HTTP server). It can be seen that the information is transferred using the TCP protocol, which enables reliability in the transfer of information between the client and the server. You can see that the server address is 127.0.0.1 and port number 30965 (as we defined it - with the last three digits being part of the end of one of our identity cards) and the client address is 127.0.0.1 and port 54068. You can see that this is a request for an PNG image file by The word GET shows that the HTTP version is 1.1 and that the client expects to receive the image file back from the server.

Server response - redirect:



Explanation: In this recording we can see the response message that the server (HTTP server) sends to the client. It can be seen that the information is transferred using the TCP protocol, which enables reliability in the transfer of information between the client and the server. You can see that the server address is 127.0.0.1 and port number 30965 (as we defined it - with the last three digits being part of the end of one of our identity cards) and the client address is 127.0.0.1 and port 54068. You can see that this is a response by The “301 Moved Permanently” status code and the HTTP version that is HTTP 1.1. In this case the server does not have the file so he sends a redirect message to the client with the address of the server that contains the image.

Client request for the PNG image file (from the image server):

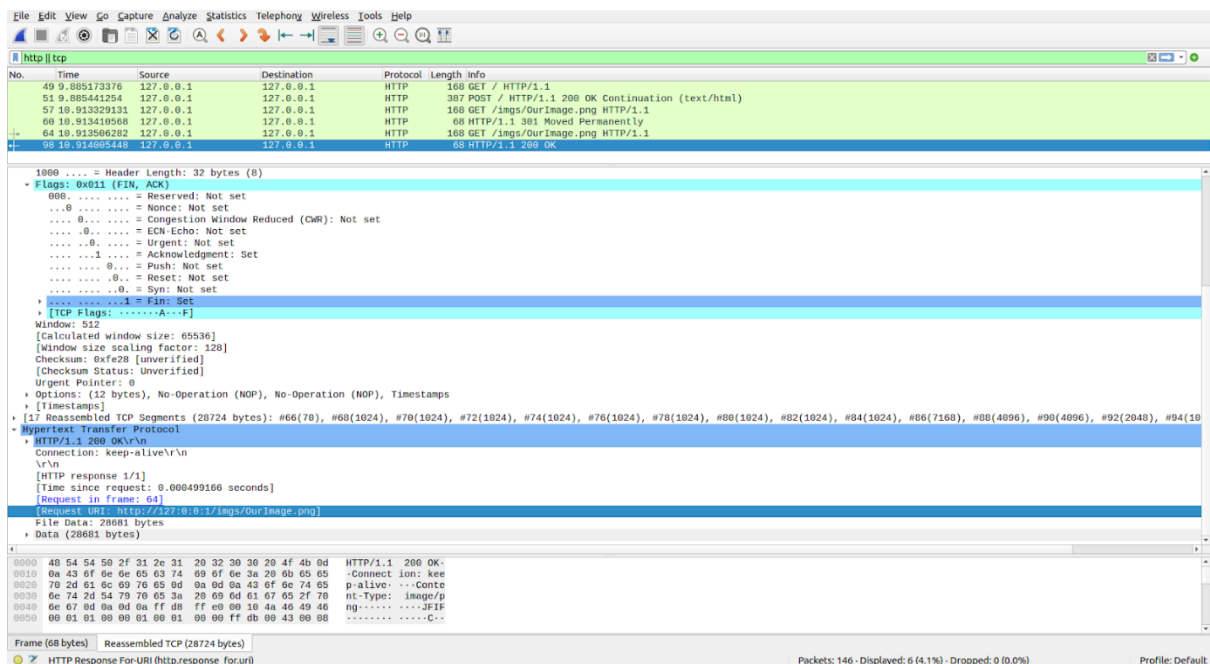
The image shows a Wireshark packet capture of an HTTP transaction. The top pane displays a list of packets. Packet 64 is selected, showing an HTTP GET request for '/imgs/OurImage.png' from 127.0.0.1 to 127.0.0.1. The middle pane shows the details of the selected packet, including the TCP header (Sequence Number: 454663382, Acknowledgment Number: 1) and the Hypertext Transfer Protocol section (GET /imgs/OurImage.png HTTP/1.1). The bottom pane shows the raw packet data in hexadecimal and ASCII.

No.	Time	Source	Destination	Protocol	Length	Info
49	9.885173376	127.0.0.1	127.0.0.1	HTTP	168	GET / HTTP/1.1
51	9.885441254	127.0.0.1	127.0.0.1	HTTP	387	POST / HTTP/1.1 200 OK Continuation (text/html)
57	19.913329131	127.0.0.1	127.0.0.1	HTTP	168	GET /imgs/OurImage.png HTTP/1.1
60	19.913410568	127.0.0.1	127.0.0.1	HTTP	68	HTTP/1.1 301 Moved Permanently
64	19.913556282	127.0.0.1	127.0.0.1	HTTP	168	GET /imgs/OurImage.png HTTP/1.1
68	19.914605448	127.0.0.1	127.0.0.1	HTTP	68	HTTP/1.1 200 OK

Sequence Number (raw): 454663382
[Next Sequence Number: 101 (relative sequence number)]
Acknowledgment Number: 1 (relative ack number)
Acknowledgment number (raw): 194336475
1000 = Header Length: 32 bytes (8)
Flags: 0x018 (PSH, ACK)
...0 = Reserved: Not set
...0 = Nonce: Not set
...0 = Congestion Window Reduced (CWR): Not set
...0 = ECH-Echo: Not set
...0 = Urgent: Not set
...1 = Acknowledgment: Set
...1 = Push: Set
...0 = Reset: Not set
...0 = Syn: Not set
...0 = Fin: Not set
[TCP Flags:AP..]
Window: 512
[Calculated window size: 65536]
[Window size scaling factor: 128]
Checksum: 6xfe8c [unverified]
[Checksum Status: Unverified]
Urgent Pointer: 0
Options: (12 bytes), No-Operation (NOP), No-Operation (NOP), Timestamps
[Timestamps]
[SEQ/ACK analysis]
TCP payload (160 bytes)
Hypertext Transfer Protocol
GET /imgs/OurImage.png HTTP/1.1
Host: 127.0.0.1
Accept: image/webp,*/*
Connection: keep-alive

Explanation: In this recording we can see the PNG image file request message that the client sends to the image server after receiving a redirected response from the HTML server. It can be seen that the information is transferred using the TCP protocol, which enables reliability in the transfer of information between the client and the server. You can see that the server address is 127.0.0.1 and port number 20630 (as we defined it - with the last three digits being part of the end of one of our identity cards) and the client address is 127.0.0.1 and port 54069 (the clients port change because he start new connection with the image server). You can see that this is a request for an PNG image file by The word GET shows that the HTTP version is 1.1 and that the client expects to receive the image file back from the server.

Image Server response - send the PNG image file:



Explanation: In this recording we can see the response message that the server (image server) sends to the client . It can be seen that the information is transferred using the TCP protocol, which enables reliability in the transfer of information between the client and the server. You can see that the server address is 127.0.0.1 and port number 20630 (as we defined it - with the last three digits being part of the end of one of our identity cards) and the client address is 127.0.0.1 and port 54069. You can see that this is a response by The “200 ok” status code and the HTTP version that is HTTP 1.1 and the type of the data that the server sends in this case PNG image.

HTTP RUDP

In Git you can also see an attempt to implement the application in RUDP. Although we were not able to complete the entire project due to lack of time, we implemented the basic components necessary for the server to function. Our work includes flow control, sliding window, Reno CC algorithm, and packet handling, which are crucial components for reliable and efficient communication between the client and server.

simulate packet loss and

To simulate packet loss, use a Linux tool called "tc" and created packet loss with the following command:

```
sudo tc qdisc change dev lo root netem loss XX%
```

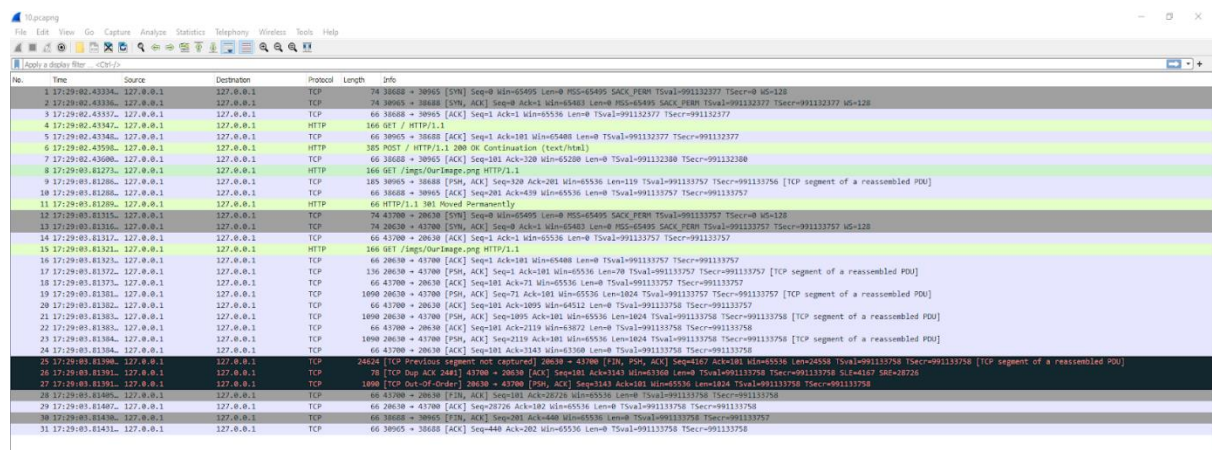
And following command:

```
sudo tc qdisc del dev lo root netem
```

To cancel packet loss.

10 percentage packets lost:

Note: These screenshots are taken from the Wireshark recordings "http_app_10.pcapng" and show the communication between the client with both servers during their connection when the packet loss percentage is 10.



The screenshot shows a Wireshark packet capture of a TCP connection. The packet list on the left shows packets 1 through 31. The packet details pane on the right shows the selected packet (No. 1) with its TCP header information. The packet bytes pane on the right shows the raw data of the selected packet. The packet list shows the following details:

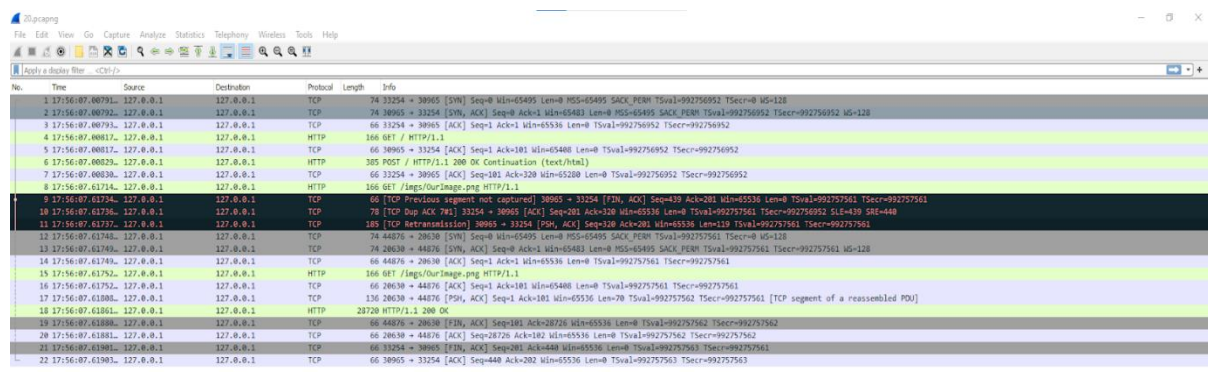
No.	Time	Source	Destination	Protocol	Length	Info
1	1.172190	192.168.1.101	192.168.1.1	TCP	74	38968 → 38965 [SYN] Seq=0 Win=65535 Len=0 MSS=65535 SACK_PERM TSval=991132377 TSecr=0 WS=128
2	1.212190	192.168.1.1	192.168.1.101	TCP	74	38965 → 38968 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=65535 SACK_PERM TSval=991132377 TSecr=991132377 WS=128
3	1.272190	192.168.1.101	192.168.1.1	TCP	66	38968 → 38965 [ACK] Seq=1 Ack=1 Win=65536 Len=0 TSval=991132377 TSecr=991132377
4	1.272190	192.168.1.101	192.168.1.1	HTTP	166	GET / HTTP/1.1
5	1.272190	192.168.1.101	192.168.1.1	TCP	66	38965 → 38968 [ACK] Seq=1 Ack=181 Win=65536 Len=0 TSval=991132377 TSecr=991132377
6	1.272190	192.168.1.101	192.168.1.1	HTTP	305	POST / HTTP/1.1 200 OK Continuation (text/html)
7	1.272190	192.168.1.101	192.168.1.1	TCP	66	38968 → 38965 [ACK] Seq=181 Ack=320 Win=65536 Len=0 TSval=991132380 TSecr=991132380
8	1.272190	192.168.1.101	192.168.1.1	HTTP	166	GET /img/OurImage.png HTTP/1.1
9	1.272190	192.168.1.101	192.168.1.1	TCP	185	38965 → 38968 [PSH, ACK] Seq=320 Ack=281 Win=65536 Len=119 TSval=991133757 TSecr=991133756 [TCP segment of a reassembled PDU]
10	1.272190	192.168.1.101	192.168.1.1	TCP	66	38968 → 38965 [ACK] Seq=281 Ack=439 Win=65536 Len=0 TSval=991133757 TSecr=991133757
11	1.272190	192.168.1.101	192.168.1.1	HTTP	66	HTTP/1.1 301 Moved Permanently
12	1.272190	192.168.1.101	192.168.1.1	TCP	74	43700 → 28638 [SYN] Seq=0 Win=65535 Len=0 MSS=65535 SACK_PERM TSval=991133757 TSecr=0 WS=128
13	1.272190	192.168.1.101	192.168.1.1	TCP	74	28638 → 43700 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=65535 SACK_PERM TSval=991133757 TSecr=991133757 WS=128
14	1.272190	192.168.1.101	192.168.1.1	TCP	66	43700 → 28638 [ACK] Seq=1 Ack=1 Win=65536 Len=0 TSval=991133757 TSecr=991133757
15	1.272190	192.168.1.101	192.168.1.1	HTTP	166	GET /img/OurImage.png HTTP/1.1
16	1.272190	192.168.1.101	192.168.1.1	TCP	66	28638 → 43700 [ACK] Seq=1 Ack=181 Win=65536 Len=0 TSval=991133757 TSecr=991133757
17	1.272190	192.168.1.101	192.168.1.1	TCP	136	28638 → 43700 [PSH, ACK] Seq=1 Ack=181 Win=65536 Len=70 TSval=991133757 TSecr=991133757 [TCP segment of a reassembled PDU]
18	1.272190	192.168.1.101	192.168.1.1	TCP	66	43700 → 28638 [ACK] Seq=181 Ack=71 Win=65536 Len=0 TSval=991133757 TSecr=991133757
19	1.272190	192.168.1.101	192.168.1.1	TCP	1850	28638 → 43700 [PSH, ACK] Seq=71 Ack=181 Win=65536 Len=1024 TSval=991133757 TSecr=991133757 [TCP segment of a reassembled PDU]
20	1.272190	192.168.1.101	192.168.1.1	TCP	66	43700 → 28638 [ACK] Seq=181 Ack=1895 Win=65536 Len=0 TSval=991133758 TSecr=991133757
21	1.272190	192.168.1.101	192.168.1.1	TCP	1890	28638 → 43700 [PSH, ACK] Seq=1895 Ack=181 Win=65536 Len=1024 TSval=991133758 TSecr=991133758 [TCP segment of a reassembled PDU]
22	1.272190	192.168.1.101	192.168.1.1	TCP	66	43700 → 28638 [ACK] Seq=181 Ack=2119 Win=65536 Len=0 TSval=991133758 TSecr=991133758
23	1.272190	192.168.1.101	192.168.1.1	TCP	1890	28638 → 43700 [PSH, ACK] Seq=2119 Ack=181 Win=65536 Len=1024 TSval=991133758 TSecr=991133758 [TCP segment of a reassembled PDU]
24	1.272190	192.168.1.101	192.168.1.1	TCP	66	43700 → 28638 [ACK] Seq=181 Ack=3143 Win=65536 Len=0 TSval=991133758 TSecr=991133758
25	1.272190	192.168.1.101	192.168.1.1	TCP	2824	[TCP Previous segment not captured] 28638 → 43700 [FIN, PSH, ACK] Seq=4167 Ack=181 Win=65536 Len=24558 TSval=991133758 TSecr=991133758 [TCP segment of a reassembled PDU]
26	1.272190	192.168.1.101	192.168.1.1	TCP	78	[TCP Dup ACK 2445] 43700 → 28638 [ACK] Seq=181 Ack=3143 Win=65536 Len=0 TSval=991133758 TSecr=991133758 SLE=4187 SRE=28726
27	1.272190	192.168.1.101	192.168.1.1	TCP	1890	[TCP Out-Of-Order] 28638 → 43700 [PSH, ACK] Seq=1143 Ack=181 Win=65536 Len=1024 TSval=991133758 TSecr=991133758
28	1.272190	192.168.1.101	192.168.1.1	TCP	66	43700 → 28638 [ACK] Seq=2826 Ack=182 Win=65536 Len=0 TSval=991133758 TSecr=991133758
29	1.272190	192.168.1.101	192.168.1.1	TCP	66	28638 → 43700 [ACK] Seq=2826 Ack=182 Win=65536 Len=0 TSval=991133758 TSecr=991133758
30	1.272190	192.168.1.101	192.168.1.1	TCP	66	38968 → 38965 [FIN, ACK] Seq=281 Ack=440 Win=65536 Len=0 TSval=991133758 TSecr=991133757
31	1.272190	192.168.1.101	192.168.1.1	TCP	66	38965 → 38968 [ACK] Seq=440 Ack=282 Win=65536 Len=0 TSval=991133758 TSecr=991133758

Explanation: From the recording we can see that all communication between the client and the servers is handled by the TCP protocol. In lines 1-3 we see the connection process between the client and the html server known as "TCP 3-way handshake" - to establish a reliable connection the Sender sends a request to connect with the Receiver (line 1 – [SYN] =synchronize) if the Receiver agrees to connect to the Sender, he sends back his confirmation (line 2 - [SYN,ACK] =synchronize & acknowledge) and then the Sender sends to the server that he has received his permission to connect (line 3 – [ACK] = acknowledge). TCP is considered a reliable protocol because it has mechanisms that allow it to track errors in transmitted packets or lost packets we mentioned before in the TCP section . As part of these mechanisms, we can see the server sending data and the client returning acknowledges [ACK] and vice versa so in fact the ACK reports that the data that was sent is correct and complete. Here, we can see the control mechanisms of the TCP protocol in action when data loss is detected. For example, we can see [TCP previous segment not captured] error (in line 25)- we receive a report about the loss of information that was not received properly, the TCP protocol makes sure that the same segment that was not sent is sent again until it informs the destination that it is complete and without any packet loss. This error can occur for a variety of reasons, including network

congestion, dropped packets, or issues with the capturing software itself. TCP numbered segments sequentially so that the receiving device can reassemble them in the correct order. When a segment is missing or dropped, the receiving device may send a request for that segment to be retransmitted. We can see another example in line 26, we get an[TCP Dup ACK] error - it can indicate that it has seen a gap in the received sequence numbers that implies the loss of one or more packets in transit. In line 27 , we get an [TCP out-of-order] error - can indicate that a particular frame was received in a different order from which it was sent. There are several reasons why TCP segments may arrive out of order, including network congestion, routing issues, or problems with the sending or receiving devices. In some cases, the out-of-order segments may be retransmissions of previously sent segments, which can further complicate the analysis.

20 percentage packets lost:

Note: These screenshots are taken from the Wireshark recordings "http_app_20.pcapng" and show the communication between the client with both servers during their connection when the packet loss percentage is 20.



The screenshot shows a Wireshark packet capture of a TCP connection. The packet list pane displays 22 packets. Packets 1-3 show the TCP 3-way handshake (SYN, SYN-ACK, ACK). Packets 4-10 show the client sending an HTTP GET request and the server responding with a 200 OK status and an HTML response. Packets 11-13 show a retransmission of the client's GET request (TCP Retransmission). Packets 14-16 show the client sending another HTTP GET request and the server responding with a 200 OK status and an HTML response. Packets 17-19 show the client sending a third HTTP GET request and the server responding with a 200 OK status and an HTML response. Packets 20-22 show the client sending a fourth HTTP GET request and the server responding with a 200 OK status and an HTML response. The packet details pane for packet 11 shows a 'TCP Retransmission' error.

No.	Time	Source	Destination	Protocol	Length	Info
1	17:56:07.00791	127.0.0.1	127.0.0.1	TCP	74	33254 → 38965 [SYN] Seq=0 Win=65535 Len=0 MSS=65535 SACK_PERM TSval=992756952 TSecr=0 WS=128
2	17:56:07.00792	127.0.0.1	127.0.0.1	TCP	74	38965 → 33254 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=65535 SACK_PERM TSval=992756952 TSecr=992756952 WS=128
3	17:56:07.00793	127.0.0.1	127.0.0.1	TCP	66	33254 → 38965 [ACK] Seq=1 Ack=1 Win=65536 Len=0 TSval=992756952 TSecr=992756952
4	17:56:07.00817	127.0.0.1	127.0.0.1	HTTP	166	GET / HTTP/1.1
5	17:56:07.00817	127.0.0.1	127.0.0.1	TCP	66	38965 → 33254 [ACK] Seq=1 Ack=181 Win=65536 Len=0 TSval=992756952 TSecr=992756952
6	17:56:07.00838	127.0.0.1	127.0.0.1	HTTP	305	200 OK (text/html)
7	17:56:07.00838	127.0.0.1	127.0.0.1	TCP	66	33254 → 38965 [ACK] Seq=181 Ack=328 Win=65536 Len=0 TSval=992756952 TSecr=992756952
8	17:56:07.61714	127.0.0.1	127.0.0.1	HTTP	166	GET /img/OurImage.png HTTP/1.1
9	17:56:07.61734	127.0.0.1	127.0.0.1	TCP	66	[TCP Previous segment not captured] 38965 → 33254 [FIN, ACK] Seq=189 Ack=281 Win=65536 Len=0 TSval=992757561 TSecr=992757561
10	17:56:07.61734	127.0.0.1	127.0.0.1	TCP	78	[TCP Dup ACK 794] 33254 → 38965 [ACK] Seq=0 Ack=281 Win=65536 Len=0 TSval=992757561 TSecr=992757561 WS=128
11	17:56:07.61737	127.0.0.1	127.0.0.1	TCP	185	[TCP Retransmission] 38965 → 33254 [PSH, ACK] Seq=328 Ack=281 Win=65536 Len=119 TSval=992757561 TSecr=992757561
12	17:56:07.61745	127.0.0.1	127.0.0.1	TCP	74	44876 → 28638 [SYN] Seq=0 Win=65535 Len=0 MSS=65535 SACK_PERM TSval=992757561 TSecr=0 WS=128
13	17:56:07.61749	127.0.0.1	127.0.0.1	TCP	74	28638 → 44876 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=65535 SACK_PERM TSval=992757561 TSecr=992757561 WS=128
14	17:56:07.61749	127.0.0.1	127.0.0.1	TCP	66	44876 → 28638 [ACK] Seq=1 Ack=1 Win=65536 Len=0 TSval=992757561 TSecr=992757561
15	17:56:07.61752	127.0.0.1	127.0.0.1	HTTP	166	GET /img/OurImage.png HTTP/1.1
16	17:56:07.61752	127.0.0.1	127.0.0.1	TCP	66	28638 → 44876 [ACK] Seq=1 Ack=181 Win=65536 Len=0 TSval=992757561 TSecr=992757561
17	17:56:07.61885	127.0.0.1	127.0.0.1	TCP	136	28638 → 44876 [PSH, ACK] Seq=1 Ack=181 Win=65536 Len=70 TSval=992757561 TSecr=992757561 [TCP segment of a reassembled PDU]
18	17:56:07.61885	127.0.0.1	127.0.0.1	HTTP	28728	HTTP/1.1 200 OK
19	17:56:07.61888	127.0.0.1	127.0.0.1	TCP	66	44876 → 28638 [FIN, ACK] Seq=381 Ack=28728 Win=65536 Len=0 TSval=992757562 TSecr=992757562
20	17:56:07.61881	127.0.0.1	127.0.0.1	TCP	66	28638 → 44876 [ACK] Seq=28728 Ack=182 Win=65536 Len=0 TSval=992757562 TSecr=992757562
21	17:56:07.61881	127.0.0.1	127.0.0.1	TCP	66	33254 → 38965 [FIN, ACK] Seq=0 Ack=448 Win=65536 Len=0 TSval=992757563 TSecr=992757563
22	17:56:07.61881	127.0.0.1	127.0.0.1	TCP	66	38965 → 33254 [ACK] Seq=448 Ack=282 Win=65536 Len=0 TSval=992757563 TSecr=992757563

Explanation: From the recording we can see that all communication between the client and the servers is handled by the TCP protocol. In lines 1-3 we see the connection process between the client and the html server known as "TCP 3-way handshake" - to establish a reliable connection the Sender sends a request to connect with the Receiver (line 1 – [SYN] =synchronize) if the Receiver agrees to connect to the Sender, he sends back his confirmation (line 2 - [SYN,ACK] =synchronize & acknowledge) and then the Sender sends to the server that he has received his permission to connect (line 3 – [ACK] = acknowledge). As we said before TCP is considered a reliable protocol because it has mechanisms that allow it to track errors in transmitted packets or lost packets we mentioned before in the TCP section . Here we can see another error that we didn't see before , in line 11 , we get an [TCP Retransmission] error - can indicates that a TCP segment has been retransmitted by the server due to an acknowledgement (ACK) not being received by the client within a certain timeframe.

In conclusion, how does the application deal with packet loss and latency issues?

This is exactly where the decision comes in why we chose to implement the application using TCP. TCP is designed to handle packet loss and delay to ensure reliable data transmission. When TCP encounters packet loss or delay, it uses various techniques to handle the situation and ensure reliable data transmission. In the case of packet loss, TCP will detect the missing packets through the use of sequence numbers and request retransmission of those packets from the sender. The sender will retransmit the missing packets, and TCP will continue to monitor for any further packet loss. When there is delay in packet transmission, TCP will use mechanisms such as congestion control and flow control to reduce the amount of data being sent, which can help alleviate network congestion and reduce the likelihood of packet loss. TCP also uses various algorithms to dynamically adjust the transmission rate based on network conditions, such as the number of retransmissions, to avoid further packet loss and improve overall network performance.

Answered questions:

Question 1

The main differences between TCP and QUIC protocols are:

1. Transport Layer Protocol: TCP is a reliable, connection-oriented transport protocol that operates at the Transport layer of the OSI model. QUIC, on the other hand, is an unreliable, connectionless transport protocol that operates at the Transport layer, but uses a user-space protocol implementation.
2. Connection Establishment: TCP requires a three-way handshake for connection establishment. QUIC, however, establishes a connection in just one round-trip time (RTT) by exchanging a single packet.
3. Congestion Control: TCP's congestion control mechanism is based on packet loss, which involves reducing the sending rate in response to packet drops. QUIC, on the other hand, uses a more advanced congestion control mechanism that is based on round-trip time and bandwidth estimation.
4. Encryption: TCP does not provide encryption by default, although it can be used in conjunction with other protocols. QUIC, on the other hand, provides encryption by default, using a variant of TLS called "0-RTT" encryption.

Question 2

CUBIC and VEGAS are two different congestion control algorithms used in TCP, where they are mainly differ by the following:

1. Approach to Congestion Control: CUBIC is a delay-based congestion control algorithm that focuses on measuring the available bandwidth by monitoring the delay in the network. It uses a cubic function to increase or decrease the sending rate based on the network conditions. VEGAS, on the other hand, is a throughput-based congestion control algorithm that focuses on estimating the available bandwidth by measuring the throughput of the network.
2. Sensitivity to Congestion: CUBIC is more sensitive to congestion than VEGAS, as it reacts more aggressively to network congestion by quickly reducing the sending rate. VEGAS, on the other hand, is less sensitive to congestion and takes a more gradual approach to congestion control. This makes VEGAS more suitable for high-bandwidth, low-delay networks, while CUBIC is better suited for high-latency networks.

Question 3

BGP (Border Gateway Protocol) is a routing protocol used to exchange routing information between different autonomous systems on the Internet. It operates at the Network layer of the OSI model and is responsible for routing packets between different networks or autonomous systems. BGP is different from OSPF (Open Shortest Path First) in several ways:

1. Scope: OSPF is an Interior Gateway Protocol (IGP) used within a single autonomous system, while BGP is an Exterior Gateway Protocol (EGP) used between different autonomous systems.
2. Path Selection: OSPF selects the shortest path to a destination based on its metric, while BGP uses a more complex path selection process that considers a variety of factors, including the AS path length, local preference, and MED (Multi-Exit Discriminator) values.
3. Metric: OSPF uses a metric based on bandwidth to determine the shortest path to a destination, while BGP uses a metric based on the number of autonomous system hops to a destination.

Therefore, BGP does not necessarily work on the shortest path between two routers. BGP selects the best path based on a variety of factors, including the AS path length and other metrics, which may not always be the shortest path. BGP is designed to provide stable and reliable routing on the Internet, and as such, it may choose longer paths to avoid congestion or other issues on the network.

Question 5

DNS (Domain Name System) and ARP (Address Resolution Protocol) are two different protocols. Overall, DNS and ARP serve different functions in computer networks where DNS is used to resolve domain names to IP addresses, while ARP is used to resolve IP addresses to MAC addresses on a local network. Furthermore, DNS operates at the Application layer of the OSI model, while ARP operates at the Data Link layer.

DNS is a connectionless protocol that uses UDP or TCP as its underlying transport protocol. ARP, on the other hand, is a connectionless protocol that uses the Address Resolution Protocol (ARP) message format.

Also, DNS clients typically cache the results of DNS queries to speed up subsequent requests, while ARP clients do not cache the results of ARP requests.

DNS resolution involves sending a query to a DNS server, which then returns the IP address associated with the domain name. ARP resolution, on the other hand, involves sending an ARP request **broadcast** to all devices on the local network, which then returns the MAC address associated with the IP address.