University of Waterloo

ECE-358 COMPUTER NETWORKS

Course Fall 2023

LAB 2

Group 151

József IVÁN GAFO (WatId: 21111635, jivangaf)

Sonia NAVAS RUTETE (WatId: 21111397, srutete)

TABLE OF CONTENTS

TASK 1 **3**
Explain Code: 3
1.Set up 3
2.Initialization 3
3. Generate the http response 5
4.Static methods 6
Screenshot of client browser: 7
Screenshots of Postman: 11
Correct GET without folder: 11
Correct GET with folder: 12
Correct HEAD without folder: 12
Correct GET with file not found error: 13
Correct HEAD with file not found error: 13
TASK 2 **14**
Client.py 14
Initialize 14
Create DNS Request 15
Send DNS request 16
Receive Response 17
Extract data from response 17
Print Response 20
Server.py 22
Setup 22
Initiate server 23
Methods to generate headers: 25
Extract data: 28
Translate individual headers into bytes 29
Screenshots of google.com 33
Screenshots of wikipedia.org 35
Task 3 **37**
1. What is Socket? 37
2. How sockets work? Why is socket required? 37
3. What are the types of Internet Sockets? Briefly explain the characteristics of each type.(Lesser known sockets are not required) 37
4. Mention at least one application of each type of socket? 38
5. Briefly explain the following function. 38
6. Draw the flow diagram of TCP Socket Programming that you used in this lab. 39
7. On which layer socket will execute in the Internet protocol stack? 40
8. Complete the following table: 40
9. What is DNS and how does it work? 40

TASK 1

Explain Code:

For task 1 we decided on the approach of creating a class WebServer, the reason is to make the code more modular, easier to maintain and easier to fix potential errors.

1.Set up

```
def __init__(self,serverIP:str,serverPort:int) -> None:
        self._server_ip=serverIP
        self._server_port=serverPort
        self._server_socket=socket(AF_INET, SOCK_STREAM)
        self._server_socket.bind((self._server_ip,self._server_port))
        self._server_socket.listen(1)
        print("The server is ready to receive")

        #Here we have the content when we don't find the requested file
        self._error_file_content="""
        <!DOCTYPE html>
        <html lang="en">
        <head>
            <meta charset="UTF-8">\r\n
            <meta name="viewport" content="width=device-width, initial-scale=1.0">\r\n
            <title>Document</title>\r\n
        </head>
        <body>\r\n
            <h1>404 Not found</h1>\r\n
        </body>
        </html>
        """
```

In the setup we open the server with an ip and port that we selected. In our case we choose the ip 127.0.0.1 and the server port 11000, and we set the server with ".bind(...)" and listen(1) from the library socket. (the meaning of this 2 functions is explained in task3) Finally we load the 404 not found error html. We decided to use a string instead of an actual html file, because we were not sure if we were allowed to submit more than 1 html file.

2.Initialization

```
def initialization(self):
        """
        This method is in charge of initializing the web server
        """
        while True:

            #We wait until we receive a request
            connectionSocket,client_address=self._server_socket.accept()
            sentence=connectionSocket.recv(2048).decode()
            print("sentence received")

            #We split the content to obtain the type of request and file path
            try:
                list_sentence = sentence.split()
                type_request = list_sentence[0]
                file_path =os.getcwd()+list_sentence[1]

                #If the type of request is not head or get we close the connection
```

```
                if type_request != "HEAD" and type_request !="GET":
                    raise ValueError("The type of request allowed is HEAD or GET")

            #In case of error we close the connection with client
            except Exception as error:
                print(f"[ERROR] An exception occurred: {error}")
                connectionSocket.close()
                continue

            #we open the path if error we return the error http response
            try:
                # We open the file name
                with open(file_path, "rb") as file:
                    file_content = file.read()
                    #We create the headers+data for the http response
                    response=self.__http_response(type_request,file_content,"200 OK",file_path)

            #If we didn't find the file we send http error response
            except FileNotFoundError:
                print("[ERROR] File not found")

                response=self.__http_response(type_request,self._error_file_content.encode(),"404 Not Found",
"ERROR")

            #If we encounter another type of error
            except Exception as error:
                print(f"[ERROR] An exception occurred: {error}")
                connectionSocket.close()
                continue

            print("sending response")
            print(response)
            #We send the response
            connectionSocket.send(response.encode())
            #We close connection with client
            connectionSocket.close()
```

We first create a loop that will iterate “Infinitely” and will handle http requests from a client. In the loop we will wait until we receive a request. When we receive the request, we will decode it and save it on the variable sentence.

We then split the sentence and we obtain a list that has all the contents of the client request.

After that, we extract the type of request and the file_path (that is the path of the actual folder+the requested path, this was done to avoid possible errors).

If the type of request is not head or get we raise a ValueError since our program was designed to only handle get and head requests. If we raise a ValueError it will be catched by the exception and we will close the connection with the client and continue to wait for another answer.

Once we have the file path and the type of request we try to open the file path. If there is no errors (it means the server successfully found the file, in our case is an html file (“helloworld.html), in this case we create an http response with the method http_response and inputs type_request, file_path, file_content and the status that in this case is “200 ok”.

In case the file is not found, we print the error on the terminal, and we generate an http response with the error content, the status code “404 Not Found” and the file path “ERROR”.

We used the file path =”ERROR” because we will use an if statement to differentiate a 200 ok and a 404 Not found error when generating a http response.

If another type of error happens when opening the file we close the connection with the client and continue to the next iteration of the loop.

After obtaining the response we encode it and we send it to the client and we close the connection with the client and we go to the next iteration to wait for another response.

3. Generate the http response

```
#Methods to generate the response of the web server
    def __http_response(self,type_request:str,file_content:bytes,status_code:str, file_path:str)->str:
        """
        Description: This method is in charge of returning the response of the http request
        @type_request: The type of request the user is requesting (HEAD or GET)
        @file_path: The file path the client is requesting
        @file_content: The content of the file the user is requesting
        @status_code: is a string containing the status code of the response
        @return: It returns the response with all the headers and possible data
        """
        #We obtain the different headers
        status="HTTP/1.1 "+status_code+"\r\n"
        connection=self.__get_connection_header()
        date=self.__get_date_header()
        server=self.__get_server_header()
        #If is an error we don't want to send the last modification date
        if(file_path=="ERROR"):
            last_mod=""
        else:
            last_mod=self.__get_last_mod_date_header(file_path)
        content_type=self.__get_content_type_header()
        #Depending if is a head or
        if type_request=="HEAD":
            content_length=self.__get_content_length_header(b"")
        else:
            content_length=self.__get_content_length_header(file_content)

        #We create the structure of the response
        response=status+date+server+last_mod+content_length+content_type+connection

        #IF is type HEAD then we only return the headers
        if type_request=="HEAD":
            return response+"\r\n"
        #If is type get we return all headers + file content
        return response+"\r\n"+file_content.decode()
```

Here we obtain all the headers (status, connection, date, server, last_modification, content_type, content_length) through static methods.

If we have a head request then the content length is 0 (since we don’t send file content) and we don’t send the file content since it is a head request.

But if is a get request then calculate the content length of the item requested by the client and we also add to the response the content of the file requested by the client.

If we get the file path “ERROR” it means that the file was not found, so we don’t include the last modification date. If we could use another html file on the submission we wouldn’t need to do this.

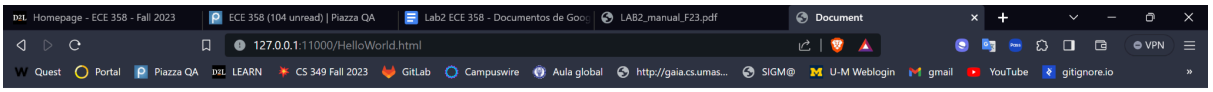
4.Static methods

```
#Static methods for obtaining headers
    @staticmethod
    def __get_date_header():
        """
        Description: Returns the actual date header
        """
        return "Date: " + datetime.datetime.now().strftime('%a, %d %b %Y %H:%M:%S GMT') + "\r\n"

    @staticmethod
    def __get_last_mod_date_header(file_path:str)->str:
        """
        Description: Return the header of the last time a file was modified
        @file_path:the file path of the file requested by client
        """
        return "Last-Modified: " + datetime.datetime.fromtimestamp(
                os.path.getmtime(file_path)).strftime('%a, %d %b %Y %H:%M:%S GMT') + "\r\n"

    @staticmethod
    def __get_content_length_header(file_content:bytes)->str:
        """
        Description: Returns the header of the content length
        @file_content: The content of the file requested by the client
        """
        return "Content-Length: " + str(len(file_content)) + "\r\n"

    @staticmethod
    def __get_server_header()->str:
        """
        Description: returns the web server software being used header
        """
        return "Server: Webserver\r\n"

    @staticmethod
    def __get_connection_header()->str:
        """
        Description: returns the connection header
        """
        return "Connection: Keep-Alive\r\n"

    @staticmethod
    def __get_content_type_header()->str:
        """
        Description: Returns the content type header of the file requested by client
        """
        return "Content-Type: text/html\r\n"
```

We decided to use static methods for almost every header because it made our code more readable, and in a hypothetical case we use this program for commercial use it would allow us to easily identify and modify a specific header.
On the date header we calculate the date using the datetime library.
For the last modification header we obtain the time of the file by using the os library.
For the content length, we do the length of the file content.
For the server header we always return Webserver, that is the name of our server.

For the connection header we return Keep-alive because our program is designed to establish persistent connections
For content type header we will always return text/html because we will always send text on an html (even if the file was not found)

Screenshot of client browser:



HAS WORKED

As we can see our webserver works because we sent the get request "http://127.0.0.1:11000/HelloWorld.html", where 127.0.0.1 is the ip and "11000" is the port of the server and "HelloWorld.html" is the file we want to access that has the text "has worked".

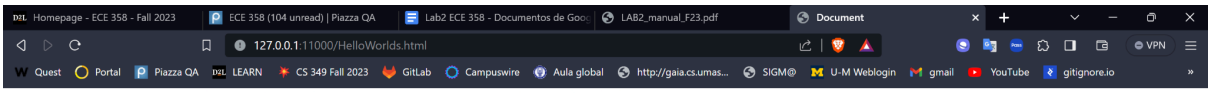
The configurations of the webserver (defining the values of ip and port #) are defined at the bottom of the webserver.py code, on the conditional if __name__=="__main__":

This is the print of the terminal on Webserver:

```
The server is ready to receive
sentence received
GET /HelloWorld.html HTTP/1.1
Host: 127.0.0.1:11000
Connection: keep-alive
Cache-Control: max-age=0
sec-ch-ua: "Brave";v="119", "Chromium";v="119", "Not?A_Brand";v="24"
sec-ch-ua-mobile: ?0
sec-ch-ua-platform: "Windows"
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/119.0.0.0 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8
Sec-GPC: 1
Accept-Language: en-GB,en
Sec-Fetch-Site: none
Sec-Fetch-Mode: navigate
Sec-Fetch-User: ?1
Sec-Fetch-Dest: document
Accept-Encoding: gzip, deflate, br
If-Modified-Since: Wed, 01 Nov 2023 20:44:49 GMT


sending response
HTTP/1.1 200 OK
Date: Wed, 08 Nov 2023 21:05:05 GMT
Server: Webserver
Last-Modified: Wed, 01 Nov 2023 20:44:49 GMT
Content-Length: 217
Content-Type: text/html
Connection: Keep-Alive

<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <h1>HAS WORKED</h1>
</body>
</html>
```

The first part we can see the get request from the browser and at the end the response we send to the browser.

This is an screenshot of the client web browser when the file is not found:



404 Not found

As we can see we want to access HelloWorlds.html (we miss type the “s” of helloworld.html) and that file doesn’t exist, so the webserver sends a 404 Not Found error.

The whole code:

```
"""
University of Waterloo Fall 2023 ECE-358 LAB-2  Group 151
József IVÁN GAFO (21111635) jivangaf@uwaterloo.ca
Sonia NAVAS RUTETE (21111397) srutete@uwaterloo.ca
V 1:0
Description: In this module we will write the code for the webserver for the task 1 of lab 2
"""
#We define the imports
import datetime
import os
from socket import *


class WebServer:
    """
    This class is in charge of running a web server for the lab 2 task 1
```

```
    """

    def __init__(self,serverIP:str,serverPort:int) -> None:
        self._server_ip=serverIP
        self._server_port=serverPort
        self._server_socket=socket(AF_INET, SOCK_STREAM)
        self._server_socket.bind((self._server_ip,self._server_port))
        self._server_socket.listen(1)
        print("The server is ready to receive")

        #Here we have the content when we don't find the requested file
        self._error_file_path=os.getcwd()+"/404_not_found.html"
        try:
            with open(self._error_file_path, "rb") as file:
                    self._error_file_content = file.read()
        except FileNotFoundError as error:
            print(f"[ERROR] An exception occurred: {error}")


    def initialization(self):
        """
        This method is in charge of initializing the web server
        """
        while True:

            #We wait until we receive a request
            connectionSocket,client_address=self._server_socket.accept()
            sentence=connectionSocket.recv(2048).decode()
            print("sentence received")

            #We split the content to obtain the type of request and file path
            try:
                list_sentence = sentence.split()
                type_request = list_sentence[0]
                file_path =os.getcwd()+list_sentence[1]

                #If the type of request is not head or get we close the connection
                if type_request != "HEAD" and type_request !="GET":
                    raise ValueError("The type of request allowed is HEAD or GET")

            #In case of error we close the connection with client
            except Exception as error:
                print(f"[ERROR] An exception occurred: {error}")
                connectionSocket.close()
                continue

            #we open the path if error we return the error http response
            try:
                # We open the file name
                with open(file_path, "rb") as file:
                    file_content = file.read()
                    #We create the headers+data for the http response
                    response=self.__http_response(type_request,file_path,file_content,"200 OK")

            #If we didn't find the file we send http error response
            except FileNotFoundError:
                print("[ERROR] File not found")
                #response=self.__http_error_response()
                response=self.__http_response(type_request,self._error_file_path,self._error_file_content,"404 Not Found")

            #If we encounter another type of error
            except Exception as error:
                print(f"[ERROR] An exception occurred: {error}")
                connectionSocket.close()
                continue

            print("sending response")
            print(response)
            #We send the response
```

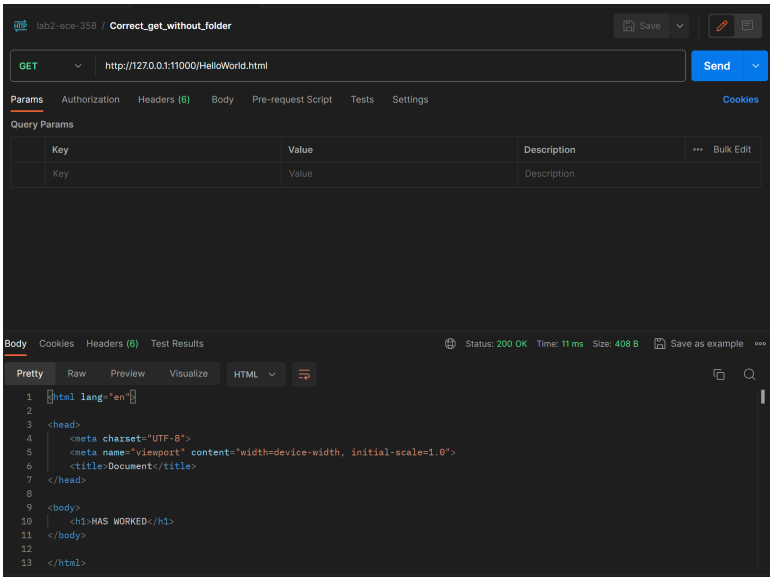
```
            connectionSocket.send(response.encode())
            #We close connection with client
            connectionSocket.close()



    #Methods to generate the response of the web server
    def __http_response(self,type_request:str,file_path:str,file_content:bytes,status_code:str)->str:
        """
        Description: This method is in charge of returning the response of the http request
        @type_request: The type of request the user is requesting (HEAD or GET)
        @file_path: The file path the client is requesting
        @file_content: The content of the file the user is requesting
        @status_code: is a string containing the status code of the response
        @return: It returns the response with all the headers and possible data
        """
        status="HTTP/1.1 "+status_code+"\r\n"
        connection=self.__get_connection_header()
        date=self.__get_date_header()
        server=self.__get_server_header()
        last_mod=self.__get_last_mod_date_header(file_path)
        content_type=self.__get_content_type_header()
        if type_request=="HEAD":
            content_length=self.__get_content_length_header(b"")
        else:
            content_length=self.__get_content_length_header(file_content)

        #We create the structure of the response
        response=status+date+server+last_mod+content_length+content_type+connection

        #IF is type HEAD then we only return the headers
        if type_request=="HEAD":
            return response+"\r\n"
        #If is type get we return all headers + file content
        return response+"\r\n"+file_content.decode()


    #Static methods for obtaining headers
    @staticmethod
    def __get_date_header():
        """
        Description: Returns the actual date header
        """
        return "Date: " + datetime.datetime.now().strftime('%a, %d %b %Y %H:%M:%S GMT') + "\r\n"

    @staticmethod
    def __get_last_mod_date_header(file_path:str)->str:
        """
        Description: Return the header of the last time a file was modified
        @file_path:the file path of the file requested by client
        """
        return "Last-Modified: " + datetime.datetime.fromtimestamp(
                os.path.getmtime(file_path)).strftime('%a, %d %b %Y %H:%M:%S GMT') + "\r\n"

    @staticmethod
    def __get_content_length_header(file_content:bytes)->str:
        """
        Description: Returns the header of the content length
        @file_content: The content of the file requested by the client
        """
        return "Content-Length: " + str(len(file_content)) + "\r\n"

    @staticmethod
    def __get_server_header()->str:
        """
        Description: returns the web server software being used header
        """
        return "Server: Webserver\r\n"
```

```python
    @staticmethod
    def __get_connection_header()->str:
        """
        Description: returns the connection header
        """
        return "Connection: Keep-Alive\r\n"

    @staticmethod
    def __get_content_type_header()->str:
        """
        Description: Returns the content type header of the file requested by client
        """
        return "Content-Type: text/html\r\n"



if __name__=="__main__":
    serverIP = "127.0.0.1"
    serverPort = 11000
    web_server=WebServer(serverIP,serverPort)
    web_server.initialization()
```

Screenshots of Postman:

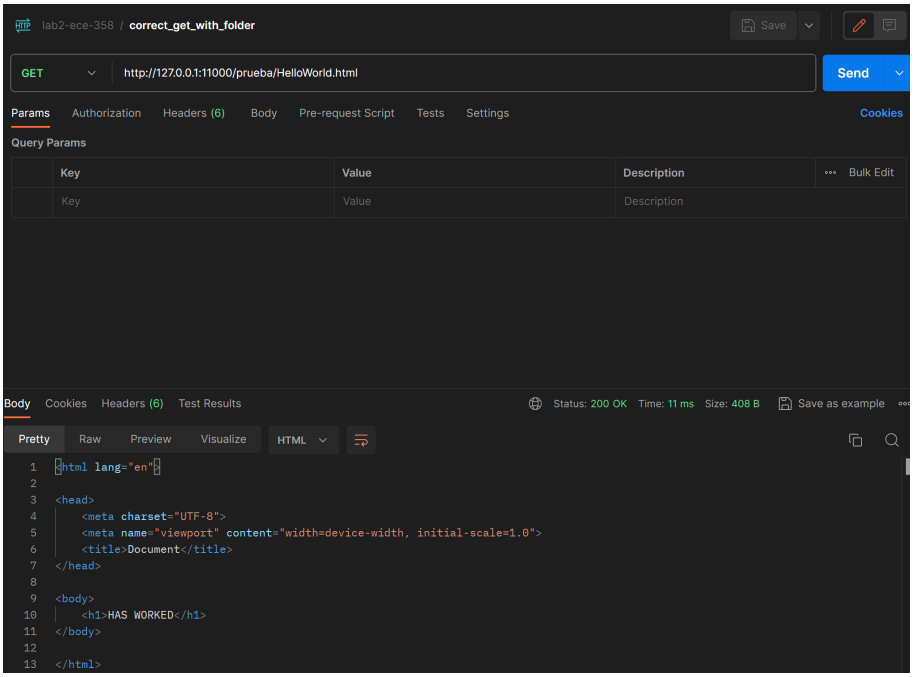
For task 1 we use Postman as a way to test our code

Correct GET without folder:

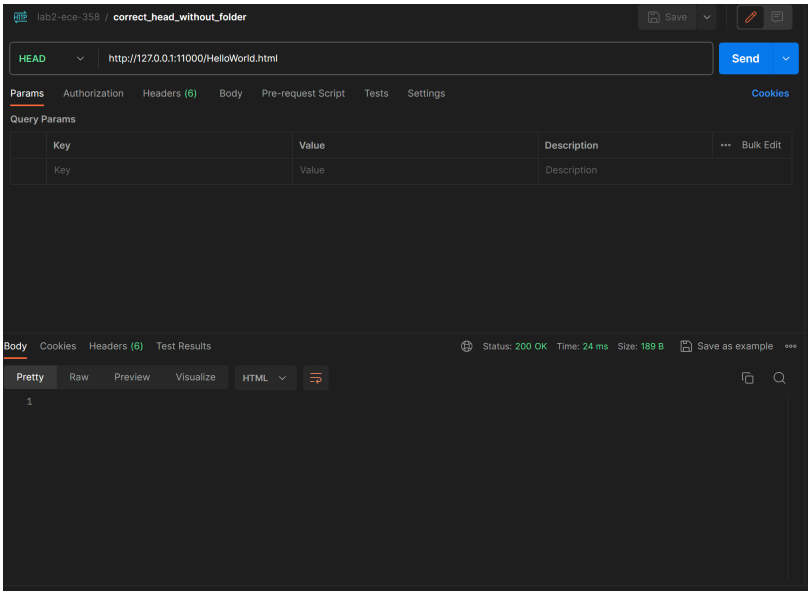


Key	Value
Date	Wed, 08 Nov 2023 20:45:23 GMT
Server	Webserver
Last-Modified	Wed, 01 Nov 2023 20:44:49 GMT
Content-Length	217
Content-Type	text/html
Connection	Keep-Alive

Correct GET with folder:

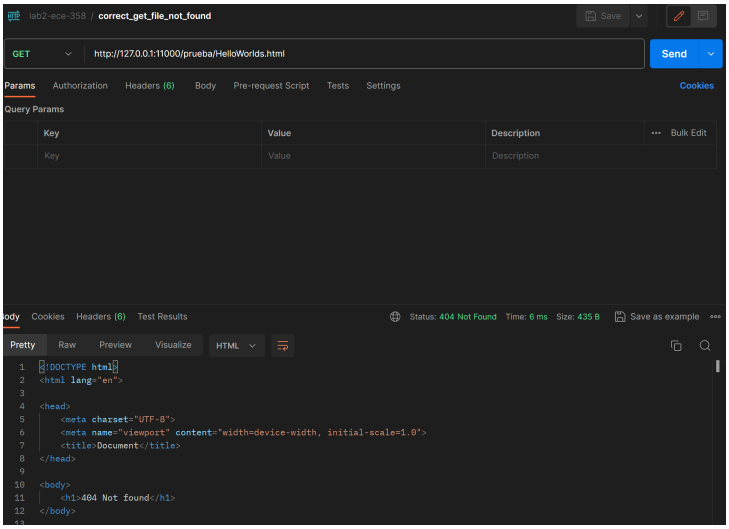


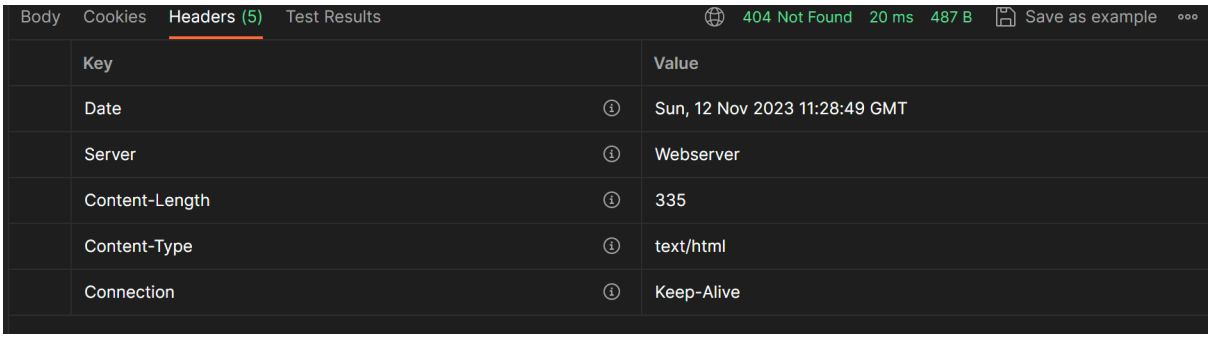
Correct HEAD without folder:



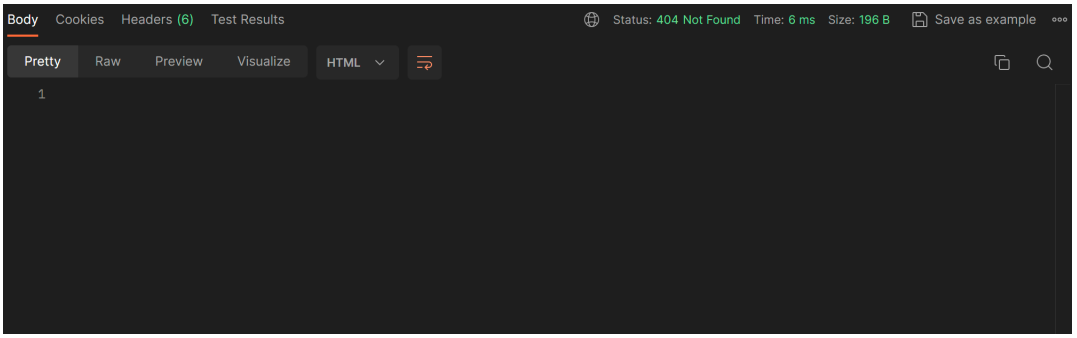
Key	Value
Date	Wed, 08 Nov 2023 20:46:45 GMT
Server	Webserver
Last-Modified	Wed, 01 Nov 2023 20:44:49 GMT
Content-Length	0
Content-Type	text/html
Connection	Keep-Alive

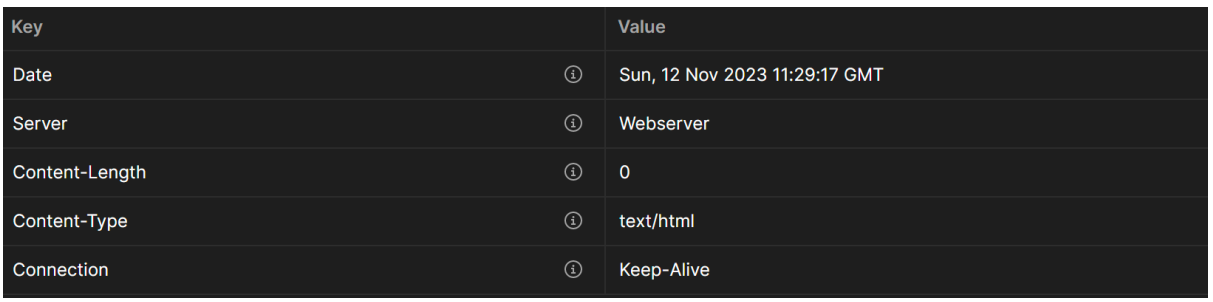
Correct GET with file not found error:





Correct HEAD with file not found error:





TASK 2

Client.py

Initialize

For creating the client we have created a Class Client. With parameters server ip and server port. Then on the init we will create a socket. After that we have just to initialize the process.

```
if __name__=="__main__":
    serverIP="127.0.0.1"
    serverPort=12000
    client=Client(serverIP,serverPort)
    client.initialize()
class Client:
    def __init__(self,server_ip:str,server_port:int) -> None:
        self.__server_ip=server_ip
        self.__server_port=server_port
        self.__client_socket=socket(AF_INET, SOCK_DGRAM)
```

The initialize function is mainly formed of a while true that will end and close the socket if the input is “end”. In this way the client can send as many requests without closing the connection.

INSIDE THE WHILE TRUE:

```
def initialize(self):
        """
        We initialize the request to the dns server
        """
        while True:
            print("Input from the user:")
            domain=input("Enter Domain Name: ")
            domain=domain.lower()

            #We finish connection if input is end
            if domain=="end":
                self.__client_socket.close()
                print("Session Ended")
                break
```

Each time we ask for the domain, we change it into lower cases to avoid sensitivity case errors. If the domain is “end”, we finish the connection, and we terminate the loop.

Create DNS Request

The dns_request is generated joining the dns headers and the dns query.

```
dns_header=self.__dns_header()
dns_query=self.__dns_query(domain)
dns_request=dns_header+dns_query
```

a. dns_header

```
def __dns_header(self)->bytes:
        """
        This method is in charge of generating the header of the dns
        """
        #We generate the ID
        random_id=random.randint(0,(2**16)-1)
        dns_id=self.int_to_bytes(random_id,2)
        #We generate the flag header
        flags = self.generate_flags()
        #We generate the other headers
        qdcount=self.int_to_bytes(1,2)#number of entries in question section
        #Based on message type
        ancount=self.int_to_bytes(0,2)#number of resource records in answer section
        nscount=self.int_to_bytes(0,2)#number of name server resource records in authorative records
        arcount=self.int_to_bytes(0,2)#number of resource records additional record section

        return dns_id+flags+qdcount+ancount+nscount+arcount
```

The dns header is formed using a function where we join the different headers: the id(generated randomly, we use 0-2^16 -1 to generate a 16 bits random id), the flags that are all generated with another function that just sets all the different values as the are predetermined (we just change the qr to 1 as it is a query). qdcount is 1 because we are doing a query for a domain. ancount,nscount and arcount are 0 value, because the client is only doing a query. The dns_header will be returned in bytes

To convert an integer to bytes we use a static method that does the work for us.

```
def generate_flags(self):
        """
        We generate the flag header
        """
        qr = "0"  # 0 is query, 1 is response
        opcode = "0000"  # Standard query
        aa = "1"    # Authoritative answer
        tc = "0"    #Message truncated
        rd = "0"    #recursion desired
        ra = "0"    #recursion avaible
        z = "000"   #for future use
        rcode ="0000"  #Response code
        flags=qr+opcode+aa+tc+rd+ra+z+rcode
```

```
        #We convert it to bytes and we return it
        return self.bits_to_bytes(flags)
```

To generate the flags of the query we used a method where we assign the values for the query. Since a query qr is 0 and we want an authoritative answer so its value is 1. After assembling the 8 bits we use a static method that is in charge of converting 8 bits into bytes.

b. The dns query

```
def __dns_query(self,domain:str)->bytes:
        """
        This method is in charge of generating the dns query
        """
        #Revise qname to bytes
        labels = domain.split(".")
        qname = b"" # Initialize
        for label in labels:
            qname += self.int_to_bytes(len(label),1)
            qname += label.encode()
        qname += self.int_to_bytes(0,1) # End of domain
        qtype=self.int_to_bytes(1,2)
        qclass=self.int_to_bytes(1,2)
        return qname+qtype+qclass
```

The dns query is created by the method dns_query, which generates it through the domain previously inserted(we have to encode each letter of the domain without taking into account the dot). Finally, the type and class has a default value of 1 for the lab2.

Send DNS request

```
self.__client_socket.sendto(dns_request,(self.__server_ip,self.__server
rt))
```

The dns_request is sent to the ip and port of the server(the socket), directly as it is already encoded in bytes.

Receive Response

```
response, addr = self.__client_socket.recvfrom(2048)
            print("Output:")
            response_dict=self.__extract_data(response.hex())
            #If there are no errors
            self.print_response(response_dict)
```

When we receive the response we extract all the information from the hexadecimal to make the print function easier and we print it.

Extract data from response

```
    def __extract_data(self,hex_data:hex)->dict:
        """
        Method of extracting the data from the server
        @hex_data: All the data of the response in hex
        @return dict: It return a dictionary with all the fields
        """
        #General data of the response
        data={
            "id_req":hex_data[:4],
            "flags_req":hex_data[4:8],
            "qdcount":hex_data[8:12],
            "ancount":hex_data[12:16],
            "nscount":hex_data[16:20],
            "arcount":hex_data[20:24],
            "question":[],
            "answers":[]
        }
        i=24
        #We extract the query data
        qdcount=int(data["qdcount"],16)
        for _ in range(qdcount):
            domain,qtype,qclass,i=self.__extract_query(hex_data,i)
            data["question"].append({
                "qname":domain,
                "qtype":qtype,
                "qclass":qclass
            })
        #We skip 8 hex that are the qsection(owner name)
        # We extract the answer section
        i+=8
        #Now we iterate as many responses we have
        ancount=int(data["ancount"], 16)
        for _ in range(ancount):
                            anname,answer_type,  answer_class,  answer_ttl,  rd_length,  rd_data,
self.__extract_answer_section(hex_data, i)
            data["answers"].append({
                "anname":anname,
                "answer_type": answer_type,
                "answer_class": answer_class,
                "answer_ttl": answer_ttl,
```

```
                "rd_length": rd_length,
                "rd_data": rd_data,
            })
        return data
```

EXTRACT DATA: First of all we create a dictionary that will be returned where we store all the received values. The reason we used a dictionary is because it allow use to easily access the information and also helps in readability of our code.

The header will be easy as we know the specific bytes. The problem comes with the question and answer part. First we are going to explain how we extract the question section in part a) and then in part b) we explain how we extract the answer from the server.

We decided to use a list in the keys “question” and “answer”, where we will append dictionaries because it makes our program capable of handling multiple queries (but for lab2 task 2 we only send 1 query) and multiple answers generated by the server.

When we extracted all the data from the answer of the server we then return the dictionary.

a. The question section:

We will go qcount times through the questions(in this lab it will always be 1). Each question is extracted through the function extract query:

```
def __extract_query(self, hex_data: hex, i: int) -> [str, hex, hex, int]:
        """
        This method is in charge of finding the fields of the query
        @hex_data: the response in hex
        @i: integer that represents the position in hex_data
        @return: returns the domain, qtype, qclass, and the position in hex_data
        """

        #obtain the first part of the domain length (*2 because they are hex not bytes)
        length_first_part=self.hex_to_int(hex_data[i:i+2])*2
        i+=2
        first_domain=self.hex_to_str(hex_data[i:i+length_first_part])
        i+=length_first_part

        #We obtain the second part (*2 because they are hex not bytes)
        length_second_part=self.hex_to_int(hex_data[i:i+2])*2
        i+=2
        second_domain=self.hex_to_str(hex_data[i:i+length_second_part])
        i+=length_second_part

        #We create the domain
        domain=first_domain+"."+second_domain
```

```
        # Skip over the null terminator
        i += 2

        return domain, hex_data[i:i+2], hex_data[i+2:i+4],i
```

This method is in charge of returning the domain(we have to read the first bits until '.' the rest until we find a 0, for each part we also have the corresponding length), the type and the class which are specific bytes.

b. The answer section:

We will go "ancount" times through the different answers as this will be the number of IP addresses for each domain. We use the function extract answer section, where we return each value the answer header, depending on its length, the only difficult one is rdata as it depends on type and class:

```
def __extract_answer_section(self, hex_data, i):
        """
        Extracts the answer section from the response in hex.
        @param hex_data: Hex data of the response.
        @param i: Position in hex_data.
        @return: Owner name, type, class, TTL, RDLength, RData, and updated position in hex_data.
        """
        #get anname
        anname=hex_data[i:i+4]
        i+=4

        #get answer type
        answer_type = hex_data[i:i+4]
        i += 4
        #GEt answer type
        answer_class = hex_data[i:i+4]
        i += 4
        #Get answer ttl
        answer_ttl = hex_data[i:i+8]
        i += 8
        #get rdlength
        rd_length = hex_data[i:i+4]
        i += 4
        hex_data_length=self.hex_to_int(rd_length)*2#*2 because they are bytes
        rd_data = hex_data[i:i + hex_data_length]
        i += hex_data_length
        return anname,answer_type, answer_class, answer_ttl, rd_length, rd_data, i
```

Print Response

Finally for printing we use the following function sending as parameter the dictionary previously returned:

```
def print_response(self,response:dict)->None:
        """
        Method to print the response from the server
        @response: Is a dictionary containing the answer of the server
        """
        #Check if we don't have any errors
        if response["flags_req"][2:]=="03":
            domain=response["question"][0]["qname"]
            print(f"[ERROR]: the DNS {domain} was not found")
        line=""
#We obtain the domain from all the data extracted
        domain=response["question"][0]["qname"]
        answer_list=response["answers"]
        for answer in answer_list:
            #We create the print message
            line=""
            line+=domain+": "
            #Get answer type
            if self.hex_to_int(answer["answer_type"])==1:
                line+="Type: A, "
            else:
                #For the moment we only accept type A answers
                raise ValueError("[ERROR]: We only accept type A")

            #get class
            if self.hex_to_int(answer["answer_class"])==1:
                line+="Class: IN, "
            else:
                raise ValueError("[ERROR]: We only accept IN")

            #get ttl
            line+="TTL: "+str(self.hex_to_int(answer["answer_ttl"]))+", "

            #get rdlength
            rdlength=self.hex_to_int(answer["rd_length"])

            if rdlength==4:
                line+="addr("+str(rdlength)+")"
            else:
                raise ValueError("[ERROR] The client only support ipv4")

            #Get ip
            ipv4=self.hex_to_ipv4(answer["rd_data"])
            line+=ipv4
            #Print the answer
            print(line)
```

- We check that there is no error, we do it by checking the flag rcode. If is “03” it means that the rcode of flags has the error 3 meaning that the domain is not found on the server.
- Then for each IP received we have to print one line: For each one, we have to print the domain, check the answer type if it is 1, print “type: A”(else an error, since for task2 we only managed type A), check the class which will be the same(if it is 1, print “class: IN”). Then prints the ttl, which will be just one of the keys of the dictionary, the length of the IP(rdlength) and the actual IP. The IP

extracted from the server is in hexadecimal so we use the following method to convert a hexadecimal into a string containing its equivalent into a string:

```
def hex_to_ipv4(hex_data:hex)->str:
    """
    Convert an hexadecimal into ipv4
    @hex_data: hexadecimal number that we want to convert into ip
    @return: It returns a string containing the ipv4
    """
    ipv4_address = ""
    aux = 0
    for i in range(4):
        ipv4_part = str(int(hex_data[aux:aux+2], 16))
        ipv4_address += ipv4_part
        if i != 3:
            ipv4_address += "."
        aux += 2
    return ipv4_address
```

Server.py

For creating the server we decided on the approach of creating a class called Server because it would make the code easier to read, easier to maintain and easy to fix errors. Our class Server is divided into 6 parts that are setup, initiate the server, generate headers, extract data, translate individual headers into bytes and translate types into bytes or hex.

Setup

```
def __init__(self,serverIP:str,serverPort:int,debug:bool=True) -> None:
    #Set properties for the server
    self.__server_ip=serverIP
    self.__server_port=serverPort

    #We load the domain records
    self.__domain_records={
        "google.com": {
            "Type": "A",
            "Class": "IN",
```

```
            "TTL": 260,
            "IP": ["192.165.1.1", "192.165.1.10"]
        },
        "youtube.com": {
            "Type": "A",
            "Class": "IN",
            "TTL": 160,
            "IP": ["192.165.1.2"]
        },
        "uwaterloo.ca": {
            "Type": "A",
            "Class": "IN",
            "TTL": 160,
            "IP": ["192.165.1.3"]
        },
        "wikipedia.org": {
            "Type": "A",
            "Class": "IN",
            "TTL": 160,
            "IP": ["192.165.1.4"]
    },
        "amazon.ca": {
            "Type": "A",
            "Class": "IN",
            "TTL": 160,
            "IP": ["192.165.1.5"]
        }
    }

    #We create the socket
    self.__server_socket=socket(AF_INET, SOCK_DGRAM)
    self.__server_socket.bind((self.__server_ip,self.__server_port))

    print("The server is ready to receive")
```

We first save the ip and port number of the server on the variables __server_ip and __server_port.

We then load the domain records where we save the data of the domains which are related to a specific ipv4.
We first thought of using a json file to store all the record domains, but then we read that we are only allowed to submit for task 2 server.py and client.py. So instead we used a dictionary where it holds the domain records.
In the final part of the server we create the socket and we bind it with the server ip and server port (explained in task 3).
Finally we print that the server is ready to receive

Initiate server

```
    #We initialize the server
    def initialize(self):
        while True:
            #Wait until we receive a request
            message, clientAddress = self.__server_socket.recvfrom(2048)


            # We extract the data and we convert it into message
            hex_message=message.hex()
            # We extract the data into a dictionary (it makes it easier to id and extract data)
            #request=self.extract_data_of_request(hex_message)
            request_dict=self.__extract_data(hex_message)


            print("Request:")
            #We print the request of the client
            self.__print_hex(hex_message)


            #We extract the relevant info from the request
            domain =request_dict["question"][0]["qname"]
            transaction_id=self.__hex_to_bytes(request_dict["id_req"])


            #We load the question section format (that is the query of the client)
            question_dict= request_dict["question"]
            dns_question=self.__generate_dns_question(question_dict)

            #Create structure of the dns answer
            dns_answer = self.__generate_answer_section(domain)


            #If is b""" it means we din't found the answer of the question asked to the client
            if dns_answer!=b"":
                #The number of ipv4 under a domain
                number_of_ipv4=len(self.__domain_records[domain]["IP"])
                dns_header=self.__generate_dns_header(transaction_id,found=True,
ancount=number_of_ipv4)#already in bytes

            else:
                #In case of dns not found ancount is 0, and the rcode is "0011", meaning the query doesn't
exist
                dns_header=self.__generate_dns_header(transaction_id,found=False,ancount=0)

            #We obtain the response in bytes to the client
            dns_response=dns_header+dns_question+dns_answer

            print("Response:")
            #We print the response
            self.__print_hex(dns_response.hex())


            #Send the answer to client
```

```
            self.__server_socket.sendto(dns_response,clientAddress)
```

For initializing the server we enter a loop while True, that will iterate until it receives an error.

We then wait for the client to send an answer. When we receive it, we store it in a message.

After that we convert the message in hexadecimals, and we extract all the query of the client into a dictionary and we save it on request_dict.

We then print the hexadecimal message by pairs of 2 on the terminal.

After that we start creating the headers for the answer, we first take from the request_dict the domain and the transaction id asked by the client.

We then generate the question header (qname,qtype,qclass) (that should be the same as the client sended) in bytes. In our program we made that is possible to have multiple questions because if in an hypothetical case that we want to extend our server functionality, the code is already implemented, that is the reason that to extract domain we use "`domain =request_dict["question"][0]["qname"]`", where ["question"] is the key to the questions on the request dictionary, [0] because for now the server is only capable of handling 1 question, but if we want to expand it we would need to create a loop, and then ["qname"] to access the domain.

We then generate the dns answer headers in bytes ((name,type,class,ttl,rdlength,rdata)*). If it returns an empty byte string it means that we didn't find an ipv4 related to the domain on the domain records of the server
If the answer header is not empty we then create the dns header with the transaction id of the client and the number of ipv4 found on domain records, with no errors.(#ancount=number_of_ipv4, rcode="0000")

If the answer header is empty, then ancount is 0 since there is no answer and rcode ="0011" meaning that we didn't find the domain on the server.

We pack all the answers on the variable dns_response, that is equal to dns_header + dns_question + dns_answer.

We then convert into hex and print by pairs on the terminal.

and finally we send back to the client the dns_response (that is in bytes) to the client.

Methods to generate headers:

```
def __generate_dns_header(self,transaction_id:bytes,found:bool, ancount: int)->bytes:
```

```
        """
        This method is in charge of generating the header of the dns
        @transaction_id: is the id of the request
        @found:it tells us if we found dns answer
        """
        #We generate the ID
        dns_id=transaction_id
            #We generate the flag header
        flags = self.__generate_flags(found)

        #We generate the other headers
        qdcount=self.__int_to_bytes(1,2)#number of entries in question section
        #Based on message type
        ancount=self.__int_to_bytes(ancount,2)#number of resource records in answer section

        nscount=self.__int_to_bytes(0,2)#number of name server resource records in authorative records

        arcount=self.__int_to_bytes(0,2)#number of resource records additional record section

        return dns_id+flags+qdcount+ancount+nscount+arcount
```

In this method we generate the dns header into bytes in function of found, that tells us if we found an ipv4 related to the domain, transaction id that tells us the id of the client query and ancount that tells us how many answers are expected on the dns response.

For the flags header we use a method called generate_flags and change depending if we found or not an ipv4.

```
def __generate_flags(self,found:bool)->bytes:
        """
        We generate the flag header
        """
        qr = "1"  # 0 is query, 1 is response
        opcode = "0000"  # Standard query
        aa = "1"    # Authoritative answer
        tc = "0"    #Message truncated
        rd = "0"    #recursion desired
        ra = "0"    #recursion available
        z = "000"   #for future use
        #If we found an ip address then is 0 (no error)
        if found:
            rcode ="0000"  #Response code
        #The name reference in the query does not exist( code 3)
        else:
            rcode="0011"

        flags=qr+opcode+aa+tc+rd+ra+z+rcode

        #We convert it to bytes and we return it
        return self.__bits_to_bytes(flags)
```

In this method we generate the header flags.
Now qr is 1 instead of 0, because now we are sending a response and not a query
If we found the ipv4 then the rcode=”0000” but if we didn’t find the ipv4 then the rcode becomes “0011” that tells us the error Name error or in other words Domain name not found.

```
def __generate_answer_section(self,domain)->bytes:
        """
        Method is in charge of creating the answer header
        @return: Returns bytes if there is a domain found else it returns None
        """

        #We first search if the domain asked exist in our database
        domain_record_dict = self.__find_domain_from_database(domain)

        #If is none it means we didn't find the question domain
        if domain_record_dict==None:
            return b""

        #We prepare the answer
        answer=b""
        #We get the list of ip
        ipv4_list=domain_record_dict.get("IP")


        #Now we will iterate the different ipv4 that are on domain_record_dict
        for ipv4 in ipv4_list:

            #name of the node (for this lab we have a fixed name of node=c0 0c)
            name =self.__int_to_bytes(192,1) + self.__int_to_bytes(12,1) #c0 0c (hex)=204(dec)

            #We obtain the conversion from
            type_code = self.__str_type_code_to_bytes(domain_record_dict.get("Type"))

            #We obtain the class of the ipv4
            class_ = self.__str_class_to_bytes(domain_record_dict.get("Class"))

            #We obtain the ttl of the ipv4
            ttl= self.__int_to_bytes(domain_record_dict.get("TTL"),4)#Time to live

            #We obtain the rdlength (we allways now is 4, since is ipv4)
            rdlength=self.__int_to_bytes(4,2)

            #We obtain the ipv4
            rdata = self.__str_ipv4_to_bytes(ipv4)

            #we obtain the ith answer
            answer += name+type_code+class_+ttl+rdlength+rdata

        return answer
```

We first search if the domain in the query exists in the domain records of the server and obtain the information of the record related to that domain (type,class,ttl and ip address).

If we didn't find a domain in our records we return an empty byte string, meaning that there is no answer generated and is up to the server to handle it on self.initialization (already explained in the previous part, on how it is handled).

If we find an ipv4 we will iterate as many times as ipv4 is under that domain.
Inside the loop we generate the name that will always be c00c for this lab, the type code, class, ttl, rdlength and rdata for every ipv4 found on domain records.

To generate type_code, class, rdata we use methods inside the class. The reason we used a method instead of putting the code there, is to have easier readability and modularity.

```
def __generate_dns_question(self, question_dict:list)->bytes:
        """
        Obtain the question section format into bytes
        @question_dict: Here we have a list of dictionaries containing the questiosn of the client
        @return: It returns the translation of the question into bytes
        """

        prev_question_query=b""
        for question in question_dict:

            #we obtain qname
            qname=self.__domain_to_bytes(question["qname"])

            #In this lab we know it has fixed values
            qtype=self.__int_to_bytes(1,2)
            qclass=self.__int_to_bytes(1,2)

            #We create a question query
            prev_question_query+=qname+qtype+qclass

        return prev_question_query
```

In this method we simply generate the dns question that should be the same as the question asked by the client.

Extract data:

```
def __extract_data(self,hex_data:hex)->dict:
        """
        Method of extracting the data from the server
        @hex_data: All the data of the response in hex
        @return dict: It return a dictionary with all the fields
        """
        #General data of the response
        data={
            "id_req":hex_data[:4],
```

```
            "flags_req":hex_data[4:8],
            "qdcount":hex_data[8:12],
            "ancount":hex_data[12:16],
            "nscount":hex_data[16:20],
            "arcount":hex_data[20:24],
            "question":[],
            "answers":[]
        }
        i=24
        #We extract the query data
        qdcount=int(data["qdcount"],16)
        for _ in range(qdcount):
            domain,qtype,qclass,i=self.__extract_query(hex_data,i)
            data["question"].append({
                "qname":domain,
                "qtype":qtype,
                "qclass":qclass
            })

        return data
```

Here we extract the data sent by the client into a dictionary (this code is similar to what we have in the client). We decided to use dictionaries because it allows us to extract information quickly and easily from the query of the client.
To extract the data from the query we first extract the fixed length header that are id_req, flags, qdcount,ancount,arcount.

To obtain a question we will iterate as many qdcount we have. For this lab it will always be 1. and inside the question we will have a list of dictionaries holding qname (containing the domain in string),qtype and qclass. To obtain this info we use the method __extract_query

```
def __find_domain_from_database(self,domain:str)->dict:
        """
        This method is in charge of finding the domain and
        returning the dictionary with the different values for that domain
        @domain: a string containing the domain we want to search
        @return: Returns a dictionary if it found the domain else it returns None
        """
        try:
            return self.__domain_records[domain]
        except Exception as error:
            print(f"[ERROR]: domain not found {error}")
            return
```

This method is in charge of searching a domain inside the database of the server. If the domain is not found we return none.

Translate individual headers into bytes

We decided to use these methods because we want to reduce the amount of code per method and also allow modularity and to have more readable code, in other words if we want to expand our code we don't have to search in the code where to modify it but simply change it on individual methods dedicated for that specific header.

```
def __str_ipv4_to_bytes(self, ipv4:str)->bytes:
        """
        This method is in charge of changing an ip string into bytes
        @ip: string containing an ip address
        @return: It returns the conversion of an ipv4 to bytes
        """
        rdata=b""
        #We iterate ipv4 without the "."
        for i in ipv4.split("."):
            rdata+=self.__int_to_bytes(int(i),1)
        return rdata
```

This method is in charge of converting an ipv4 address into bytes. For that we first split the ip to have a list of string ip and then we slowly convert it into bytes and store it in rdata and finally we return it.

```
def __str_class_to_bytes(self, class_str:str)->bytes:
        """
        Method in charge of translating an str class into its equivalent code in bytes
        @class_str: the class that we want to translate into bytes code
        @return: We return the class into bytes
        """
        if class_str=="IN":
            class_bytes=self.__int_to_bytes(1,2)#1=class IN(internet)
        else:
            raise ValueError("[ERROR] We only accept type IN for this lab")
        return class_bytes
```

This method is in charge of translating a class string into a byte code (e.g the class IN is 1 in bytes)

```
def __str_type_code_to_bytes(self, type_code_str:str)->bytes:
        """
        Method in charge of translating a string type code into bytes
        @type_code_str: A string containing the type code (e.g A)
        @return: It return the type code translated to bytes
        """
        type_code_bytes=b""
        if type_code_str=="A":
            type_code_bytes=self.__int_to_bytes(1,2)#1=type A
```

```
        else:
            raise ValueError("[ERROR] we only accept type A for this lab")
        return type_code_bytes
```

This method is in charge of translating the type code string into its equivalent number in bytes (e.g type A is 1 in bytes)

```
def __domain_to_bytes(self,domain:str)->bytes:
        """
        Method to convert a string containing a domain into bytes
        @domain: string containing the domain
        @return: the domain converted into bytes
        """
        #Variables
        qname = b"" # Initialize
        domain_split = domain.split(".")

        #We transform the domain into bytes
        for part_domain in domain_split:
            #obtain the length
            qname_length=self.__int_to_bytes(len(part_domain),1)
            #Obtain the part of domain
            part_domain_bytes=part_domain.encode()

            qname += qname_length+part_domain_bytes

        #End of domain
        qname += self.__int_to_bytes(0,1) # End of domain
        return qname
```

This method is in charge of converting a domain into its equivalent byte string. We first use split, to have a list of strings e.g google.com=[“google”,”com”]
We then iterate that list where we first put the length of the domain and then the domain translated to bytes and then we repeated on the next iteration.
At the end we add a 0 byte meaning that qname finished and we return it

Static Methods

We used static methods because it allow us to translate easier and without errors different types (such as bits to bytes,hex to int)

```
def __int_to_bytes(number:int,byte_size:int)->bytes:
        """
        Method to convert an integer to bytes
        @number: The number we want to convert into bytes
```

```
        @byte_size: How many bytes do we want to generate
        @return bytes: we return the conversion of number into bytes
        """
        #Byteorder big= most significant byte comes first
        return number.to_bytes(byte_size,byteorder="big")

    @staticmethod
    def __bits_to_bytes(bits:str)->bytes:
        """
        Method in charge of translating bits into bytes
        @bits: A string containing bits
        @return bytes: We return the conversion of bits to bytes
        """
        if len(bits)%8!=0:
            raise ValueError("[Error] The bit length must be multiple of 8")

        #We divide the bits by chunks of 8
        byte_chunks=[bits[i:i+8] for i in range(0, len(bits), 8)]

        result_in_bytes = bytes([int(chunk, 2) for chunk in byte_chunks])
        return result_in_bytes

    @staticmethod
    def __hex_to_bytes(hex_data:hex)->bytes:
        """
        Method to translate hex into bytes
        @hex_data: data in hex to translate into bytes
        @return: The translation of hex into bytes
        """
        return bytes.fromhex(hex_data)

    @staticmethod
    def __hex_to_int(hex_data:hex)->int:
        """
        Convert a hexadecimal number to an integer
        @hex_data: hexadecimal numbers that contain an integer
        @return str: return te conversion from hex to in
        """
        return int(hex_data,16)

    @staticmethod
    def __hex_to_str(hex_data:hex)->str:
        """
        Method to convert a hexadecimal into a string
        @hex_data: hexadecimal numbers that contain a string
        @return str: return te conversion from hex to string
        """

        return bytes.fromhex(hex_data).decode('utf-8')
```

Most functions are already python functions with the exception of bits to bytes.

How this method works is that we have a string of length module 8 and we want to convert it into bytes.
First we divide the string into byte chunks of length , we then convert each byte chunk into its equivalent byte number and we return the result of all the bits.

```python
@staticmethod
    def __print_hex(hex_number:hex)->None:
        """
        Method in charge of printing hex numbers
        @hex_data: The data we want to print
        """
        # 16 numbers are equivalent to 32 characters
        for i in range(0, len(hex_number), 32):
            group = hex_number[i:i + 32]

            # Join pairs of hex digits with a space for better readability
            formatted_group = ' '.join(group[i:i + 2] for i in range(0, 32, 2))

            # Print the formatted group
            print(formatted_group)
```

This method is in charge of printing a hex string by pairs of 2. that we see in the terminal of the server.

Screenshots of google.com

DNS query:

3b 75 04 00 00 01 00 00 00 00 00 00 06 67 6f 6f
67 6c 65 03 63 6f 6d 00 00 01 00 01

TYPE	KEY	VALUE
DNS HEADER	ID	3b 75
	FLAGS	04 00
	QCOUNT	00 01
	ANCOUNT	00 00

	NSCOUNT	00 00
	ARCOUNT	00 00
QUERY	QNAME	06 67 6f 6f 67 6c 65 03 63 6f 6d 00
	QTYPE	00 01
	QCLASS	00 01

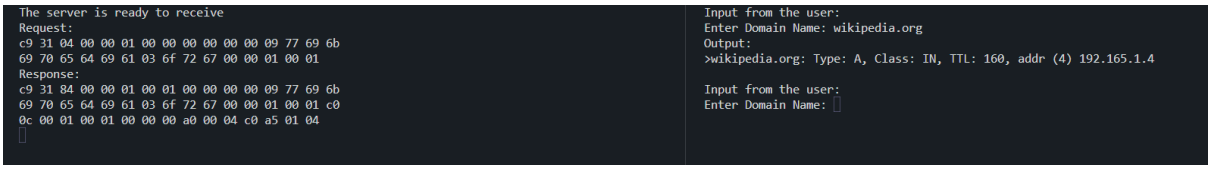
DNS response:

3b 75 84 00 00 01 00 02 00 00 00 00 06 67 6f 6f
67 6c 65 03 63 6f 6d 00 00 01 00 01 c0 0c 00 01
00 01 00 00 01 04 00 04 c0 a5 01 01 c0 0c 00 01
00 01 00 00 01 04 00 04 c0 a5 01 0a

TYPE	KEY	VALUE
DNS HEADER	ID	3b 75
	FLAGS	84 00
	QCOUNT	00 01
	ANCOUNT	00 02
	NSCOUNT	00 00
	ARCOUNT	00 00
QUERY	QNAME	06 67 6f 6f 67 6c 65 03 63 6f 6d 00
	QTYPE	00 01
	QCLASS	00 01
ANSWER 1	NAME	c0 0c
	TYPE	00 01
	CLASS	00 01
	TTL	00 00 01 04
	RDLENGTH	00 04
	RDATA	c0 a5 01 01
ANSWER 2	NAME	c0 0c

	TYPE	00 01
	CLASS	00 01
	TTL	00 00 01 04
	RDLENGTH	00 04
	RDATA	c0 a5 01 0a

Screenshots of wikipedia.org



DNS query:

c9 31 04 00 00 01 00 00 00 00 00 00 09 77 69 6b
69 70 65 64 69 61 03 6f 72 67 00 00 01 00 01

TYPE	KEY	VALUE
DNS HEADER	ID	c9 31
	FLAGS	04 00
	QCOUNT	00 01
	ANCOUNT	00 00
	NSCOUNT	00 00
	ARCOUNT	00 00
QUERY	QNAME	09 77 69 6b 69 70 65 64 69 61 03 6f 72 67 00
	QTYPE	00 01
	QCLASS	00 01

DNS Response:

c9 31 84 00 00 01 00 01 00 00 00 00 09 77 69 6b
69 70 65 64 69 61 03 6f 72 67 00 00 01 00 01 c0
0c 00 01 00 01 00 00 00 a0 00 04 c0 a5 01 04

TYPE	KEY	VALUE
DNS HEADER	ID	c9 31
	FLAGS	84 00
	QCOUNT	00 01
	ANCOUNT	00 01
	NSCOUNT	00 00
	ARCOUNT	00 00
QUERY	QNAME	09 77 69 6b 69 70 65 64 69 61 03 6f 72 67 00
	QTYPE	00 01
	QCLASS	00 01
ANSWER	NAME	c0 0c
	TYPE	00 01
	CLASS	00 01
	TTL	00 00 00 a0
	RDLENGTH	00 04
	RDATA	c0 a5 01 04

Task 3

1. What is Socket?

It is a UNIX interface for remote process communication and an abstract data structure as an endpoint of a communication link that supports full duplex information exchange (the data can be sent and received simultaneously), using it widely as a primitive for communication over the Internet. The protocols supported are TCP and UDP.
In socket programming it will be a door between the application process and end-end-transport layer.

2. How sockets work? Why is socket required?

A socket is an endpoint of communication, which means it is a door between application process and end-end-transport protocol. It works in the following way:

1. There has to be a bind server socket and a client socket.
2. The server has to start listening and the client has to connect to it
3. Once they are connected, at the moment the client generate a request and send it to the server, this one will generate a response.
4. The response is sent again to the client, which will extract the information needed.
5. Finally, it will close the connection.

Sockets are required for multiple reasons such as, the abstraction of network complexity, the possibility of selecting a different protocol, the ability of full-duplex communication and the cross-platforms(availability of multiple Operating Systems).

3. What are the types of Internet Sockets? Briefly explain the characteristics of each type.(Lesser known sockets are not required)

The types are:

-Stream sockets(uses TCP protocol as default), which characteristics are: reliable(packets received in order), connection-oriented communication and less efficient.

-Datagram sockets(uses UDP protocol as default), it is unreliable(packets are received out of order), connectionless communication and more efficient.

4. **Mention at least one application of each type of socket?**

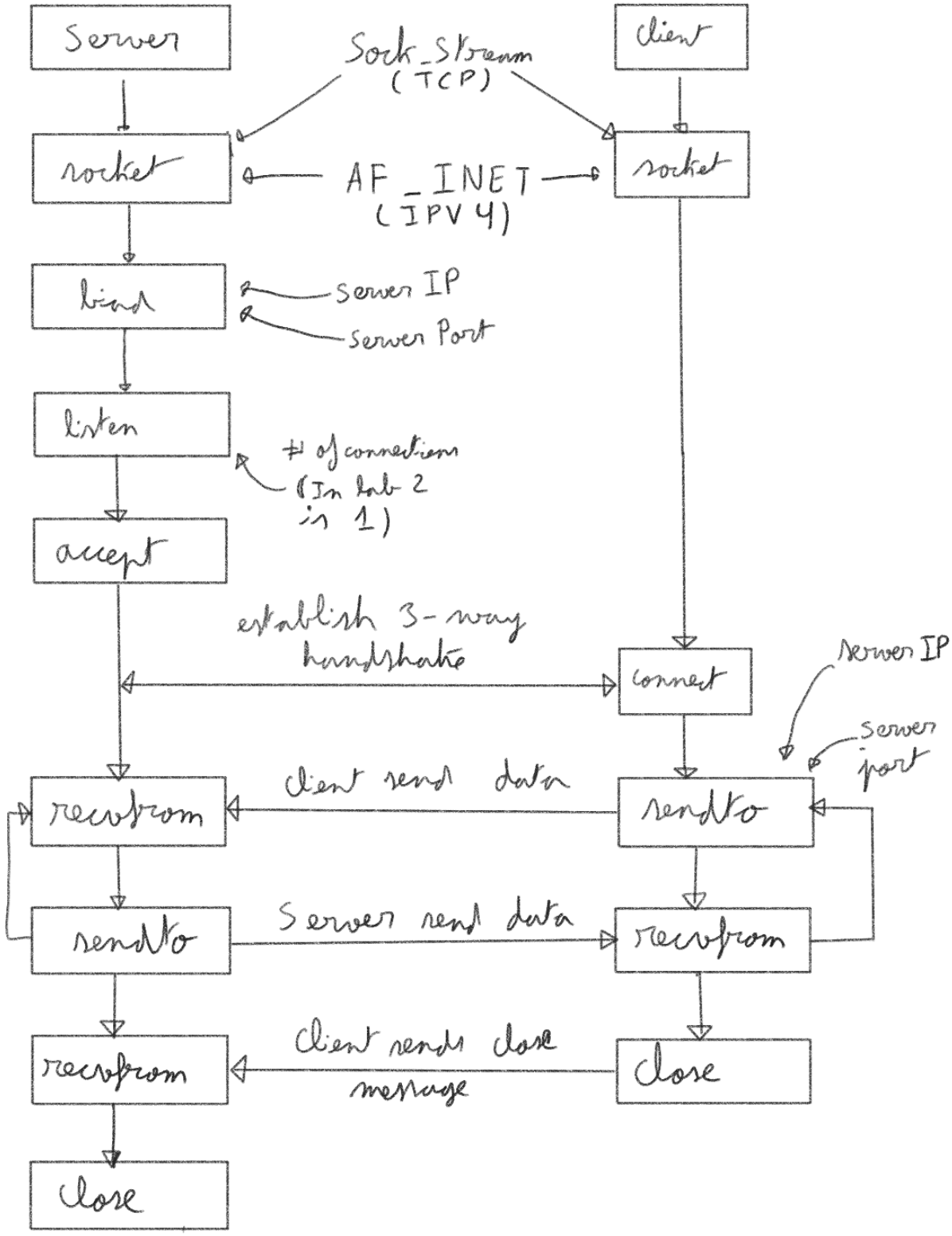
A stream socket's application could be web browsing as it is reliable, receiving the packets in order, which means we will always have all the needed information. For web browsing probably is not the most efficient way but it will be the most reliable, we ensure the data over efficiency.

A datagram socket's application could be streaming media on real time as it is really efficient, we need all the data in real-time, continuously. Probably the received information is not reliable, as we could lose information in some way because of receiving the packets out of order. But the point is having the information on time, that's why we should use this type of socket when it is required to have continuous data, e.g streaming services.

5. **Briefly explain the following function.**

FUNCTION	DESCRIPTION
socket()	It creates a new socket and initializes it using its parameters(ex.for an internet socket, choosing the type of socket), it is needed for listening or using that specific socket.
bind()	It assigns a name to the socket. Normally, we bind the Ip and the port to give it a name.
connect()	Initiate a connection with the specified socket name
listen()	Prepare the socket to start accepting connections, establish a queue for connection. Its parameter will be how many connections in the queue we can handle.
accept()	Extracts a connection from the queue
close()	Close the socket

6. **Draw the flow diagram of TCP Socket Programming that you used in this lab.**



7. **On which layer socket will execute in the Internet protocol stack?**

   The TCP socket is working at the transport layer of the internet protocol stack. Because the TCP socket is in charge of demultiplex or multiplex data to/from the application layer. e.g in task 1 we are multiplexing the information of the application (the html file) and then we send it to the TCP socket that will multiplex other headers such as the source port and destination port.

8. **Complete the following table:**

PROTOCOL	PORT #	COMMON FUNCTION
HTTP	80	enables world wide web
FTP	20,21	transfer files
IMAP4	143	retrieve email
SMTP	25	send email
Telnet	23	remote login
POP3	110	receive email

9. **What is DNS and how does it work?**

   DNS stands for Domain Name System and is a very important protocol for the internet.

   Its main function is to allow users to search with human information (domain name) and obtain its equivalent IP address. In other words a DNS server works as a phone book. When we want to know the phone number of a friend we first search for the name (domain name) and once we find it, we have a telephone number (IP address) associated with that name and now we can call that friend (establish the connection to the server with that IP address).

   First the client will enter a domain name, the computer will then search the related Ip address to that domain locally. If not found it will use the “contact of last resort”, in other words it will contact the root DNS server, then it will

contact a top level domain (TLD) server, that then it will finally go to the authoritative DNS server where we will find the ip address for that domain.

e.g I want to know the IP address of “lidl.hu”, first I will contact the local DNS that will be the University of waterloo, the local DNS server will search for that domain, if not found it will ask the root DNS server, that will ask the TLD server in this case “.hu DNS server” and then it will ask the authoritative DNS server in this case lidl.hu and we would obtain the ip related to that domain (lidl.hu./104.103.187.17).
And now the client can establish a connection to that server.