ECE 358 Lab 2: Socket Programming Using Python

Andrew Zhao, Orson Marmon

Table of Contents:

Task 1

Explain Your Code - Source Code Available Here:
In the 'server' function we follow the typical setup when creating a server. First an AF_INET, SOCK_STREAM (TCP) socket is created, then we bind the socket to a port of our choosing, and lastly listen on that port. Once the server is set up and ready to listen, we enter an infinite while loop that accepts connections. Once a connection has been accepted we parse the HTTP Request and subsequently create a HTTP Response to send back to the accepted connection. Finally, we close the connection.

```
def server():
    # 1. socket
    # 2. bind
    # 3. listen
    # 4. accept
    # 5. read
    # 6. write
    # 7. close
    serverSocket = socket(AF_INET,SOCK_STREAM)
    serverSocket.bind((SERVER_IP, SERVER_PORT))
    serverSocket.listen(1)
    print ("The server is ready to receive")
    while True:
        connectionSocket, addr = serverSocket.accept()
        http_request = connectionSocket.recv(2048).decode()

        http_response = ""
        if http_request[0:3] == "GET":
            print('GET Request')
            http_response = parse_http_request(http_request, RequestType.GET)
        elif http_request[0:4] == "HEAD":
            print('HEAD Request')
            http_response = parse_http_request(http_request, RequestType.HEAD)


        connectionSocket.send(http_response.encode())
        connectionSocket.close()
```

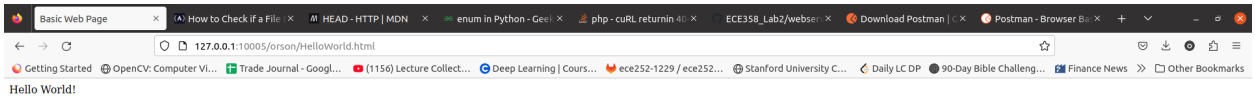
The ‘generate_response_header’ function creates the appropriate HTTP Response header to send back in the response. The connection, date, server, last-modified, content-length, and content-type field are all set. If the file is not found (404 Not Found) the content-length is set to 0 and the last-modified field of the message is set to the date of the request.

```
def generate_reponse_header(request):
    # Header Contains the following fields:
    # 1. Connection
    # 2. Date
    # 3. Server
    # 4. Last-Modified
    # 5. Content-Length
    # 6. Content-Type
    print('Generating Response Header')

    # use time now for time message request was created as
    # there is virtually no delay of any sorts since client is running from local source
    utc_date = str(formatdate(timeval=None, localtime=False, usegmt=True))
    date = "Date: " + utc_date + CR_LF

    server_name = "Server:" + SERVER_NAME + CR_LF
    connection = "Connection:" + "keep-alive" + CR_LF
    content_type = "Content-Type:" + "text/html" + CR_LF
    content_length = "Content-Length:" + str(0) + CR_LF # if file not found content length will remain 0 - ERROR 404
    last_modified = "Last-Modified:" + utc_date + CR_LF# if file not found date last modified will be trivially set to date of request

    # https://stackoverflow.com/questions/1321878/how-to-prevent-favicon-ico-requests
    # certain browsers send a request for favicon.ico but it does not always exist

    file_name = get_file_name(request)

    if not os.path.isfile(file_name):
        print("ERROR 404")
    else:

        file_name = get_file_name(request)
        content_length = "Content-Length:"  + str(os.stat(file_name).st_size) + CR_LF # file size in bytes

        # get time file was last modified
        # RFC 1123 format needed for datetime
        modify_time = os.path.getmtime(file_name)
        modify_date = datetime.datetime.fromtimestamp(modify_time)
        modify_date_utc = modify_date.astimezone(datetime.timezone.utc)
        last_modified = "Last-Modified:" + str(format_datetime(modify_date_utc, usegmt=True)) + CR_LF




    # construct header
    header_lines = date + server_name + last_modified + content_length + content_type + connection + CR_LF

    #print("HTTP Header Response: \n{}".format(header_lines))

    return header_lines
```

The ‘parse_http_request’ function parses the http request and determines if a GET or HEAD request was made. If the file is found the status line for the HTTP Response message is set accordingly (either ‘200 OK’ or ‘404 Not Found’). The header is then generated and the response is constructed conforming to the HTTP Response message format.

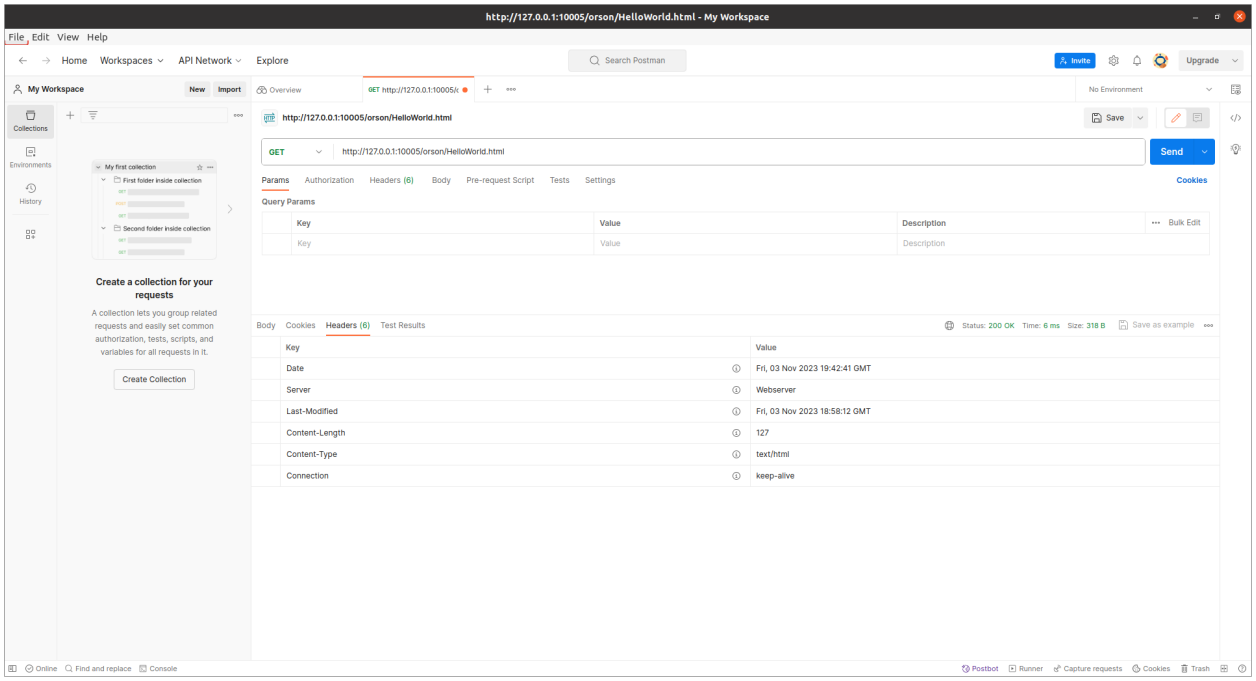
```
def parse_http_request(request, type):
    status_line = ""
    file_contents = ""
    # if file not found send error 404
    if not os.path.isfile(get_file_name(request)):
        status_line = "HTTP/1.1 404 Not Found" + CR_LF
    else:
        status_line = "HTTP/1.1 200 OK" + CR_LF
        file_contents = parse_file(get_file_name(request))

    header_lines = generate_reponse_header(request)

    file_contents_to_send = file_contents if type == RequestType.GET else ""

    response = status_line + header_lines + file_contents_to_send

    return response
```

Screenshot of Client Browser:

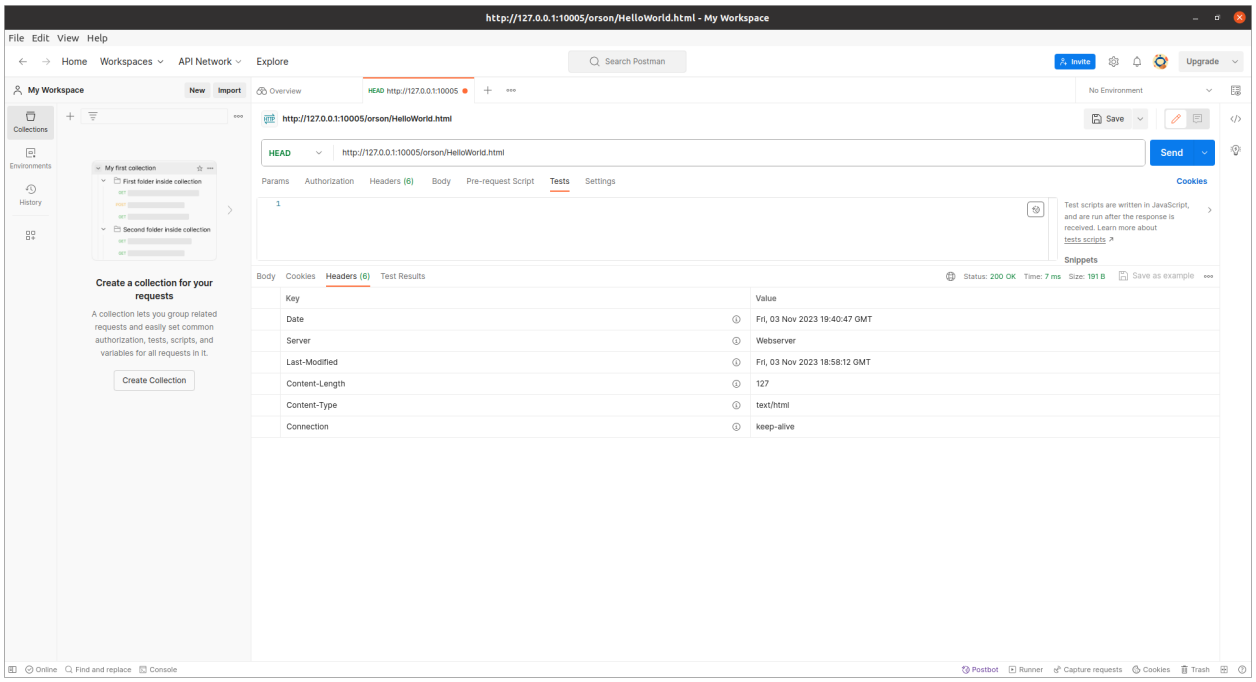


Screenshot of PostMan for GET and HEAD Request:

Get:



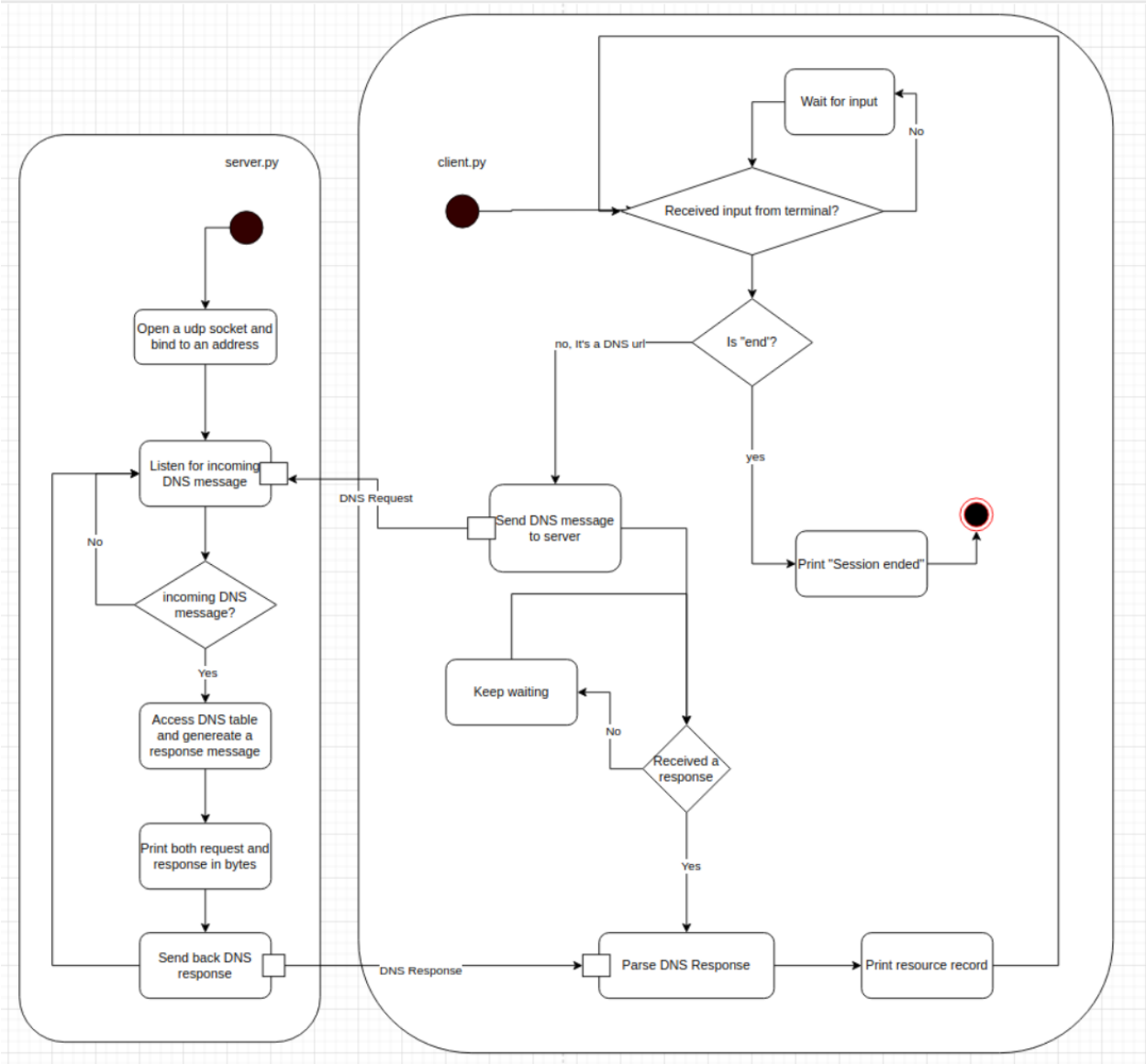
HEAD:



Task 2

There are two separate programs, a server.py and a client.py. Server.py and Client.py communicate using sockets. The control flow for the programs are as follows. Assume big endian for the entire question.

UML Activity Diagram of DNS Client-Server



Server urls are held in a dictionary with the url as the key and a list of the values for the resource record as the values that correspond to the keys. Whenever constructing a DNS message, helper functions are used to help abstract away the details. In addition, there are comments throughout the code that explain exactly what each line of code does.

The client makes an initial DNS request through the help of two helper functions, makeDNSHeader() and makeDNSBody() which combine to make the header and question. The code for the two are as follows.

Code snippet for making the DNS header and DNS question

```
def makeDNSHeader():
    DNS_response_header = bytearray()
    #ID
    random_bytes = os.urandom(2)
    DNS_response_header.extend(random_bytes)
    #Flags
    DNS_response_header.extend((0b0000010000000000).to_bytes(2, 'big'))
    #QDCount
    DNS_response_header.extend((0x0001).to_bytes(2, 'big'))
    #ANCOUNT
    DNS_response_header.extend((0x0000).to_bytes(2, 'big'))
    #NSCOUNT
    DNS_response_header.extend((0x0000).to_bytes(2, 'big'))
    #ARCOUNT
    DNS_response_header.extend((0x0000).to_bytes(2, 'big'))
    return DNS_response_header

def makeDNSBody(url):
    DNS_request_body = bytearray()
    #QNAME
    qname = bytearray()
    # url is split on .
    url_array = url.split('.')
    for label in url_array:
        # a section is www or google or com

        qname.extend(len(label).to_bytes(1, 'big'))
        for letter in label:
            qname.extend(ord(letter).to_bytes(1, 'big'))
    qname.extend((0x00).to_bytes(1, 'big'))
    DNS_request_body.extend(qname)
    #QTYPE
    DNS_request_body.extend((0x0001).to_bytes(2, 'big'))
    #QCLASS
    DNS_request_body.extend((0x0001).to_bytes(2, 'big'))

    return DNS_request_body
```

The header is constructed by starting with a random 2 byte sequence to generate the ID followed by the flags which are already given in the manual, QR is set to 0 since this is a query while every other flag is predetermined. QDCOUNT, ANCOUNT, NSCOUNT and ARCOUNT are already pregiven and thus are hardcoded for the header.

The body or question is a bit more complicated. To generate QNAME, the input string is first assumed as valid since the manual says so. The string is then split on the '.' character since every input is given as a series of labels separated by '.'. For example, google.com is the label "google" and then the label "com" while being separated by a '.' character. Before each label, the length of the label is needed for the QNAME. Thus, the size of the label is first added to the bytearray before adding each letter's ASCII value to the byte array after. The null character is immediately added at the end of QNAME. QTYPE and QCLASS are both given and can be hardcoded to be 0x0001. The query is then sent to the server which parses it for a response.

As the server parses the request, it generates the response simultaneously. This creates one difficulty as the header for the response has a field called ANCOUNT which can only be known once the url of the request is parsed. Thus, the body of the request is first read before the header of the request to obtain this url first. The body itself is composed of the question from the request and answers parsed from the request. The following code is for creating the answers of the DNS response.

Code snippet for DNS response answers

```
def getDNSAnswers(request_message):
    # Create a DNS answer
    DNS_response_answers = []

    # unpack question
    request_payload = request_message[12:]

    # remove the QType and QClass or the last 4 bytes
    request_payload_name = request_payload[:-4]
    url = get_url(request_payload_name)
    DNS_records = DNS_table[url.lower()]

    # generate response body
    DNS_response_answer = bytearray()

    # NAME
    DNS_response_answer.extend((0xc00c).to_bytes(2, 'big'))

    # TYPE
    DNS_response_answer.extend((0x0001).to_bytes(2, 'big'))

    # CLASS
    DNS_response_answer.extend((0x0001).to_bytes(2, 'big'))

    # TTL
    DNS_response_answer.extend((DNS_records[2]).to_bytes(4, 'big'))

    # RDLENGTH
    # we know the rdlength as 4
    DNS_response_answer.extend((0x0004).to_bytes(2, 'big'))

    for i in range(len(DNS_records[3])):
        DNS_response_answers.append(DNS_response_answer.copy())

    for i in range(len(DNS_records[3])):
        byte_ip_address = bytearray()
        # RDATA
        ip_address_array = DNS_records[3][i].split('.')

        for string in ip_address_array:
            #add each portion of the ip address as a byte
            byte_ip_address.extend((int(string)).to_bytes(1, 'big'))


        DNS_response_answers[i].extend(byte_ip_address)

    return DNS_response_answers, url
```

When the request message is passed, the header is first removed to parse the body. The header is known to be 12 bytes and thus is an easy operation. Next, both QTYPE and QCLASS are 2 bytes each and are known to be at the end of the bytearray and can be sliced off

too. This gives the QNAME and is passed to a helper function called get_url(). Before explaining get_url(), the assumption that it returns a valid string is made to explain the rest of getDNSAnswers(). The url is then set to lower case to handle for uppercase and passed into the DNS dictionary to return the record. The dictionary has index 0 as the TYPE, index 1 as the CLASS, index 2 as the TTL and index 3 as the resources themselves. There can be multiple answers for the same request and thus multiple answers can be returned. However, each answer has the same NAME, TYPE, CLASS, TTL and RDLENGTH and are all hardcoded. Since python arrays are by reference, the bytearray used to create this shared answer has to be shallowly copied based on the number of resource records found.

RDATA is a sequence of 4 bytes since ip addresses are always 4 bytes long. The DNS dictionary stores each IP address as a string with '.' characters to separate each byte. The IP address is first split on the '.' character because of that reason. Next, each of the split strings are then converted to their numeric values before being appended to the byte array. This is done for each resource found in the DNS dictionary. When RDATA is finished parsing for every resource record found, the answer(s) are returned along with the url.

As mentioned earlier the url is parsed from QNAME. The code is as follows:

Code Snippet for Getting URL from QNAME

```
def get_url(QNAME: bytearray):
    # parse the octal request to get a string

    index = 0
    url_name = ""
    while True:
        # get the byte
        label_length = QNAME[index]
        index += 1
        if(label_length == 0):
            break
        for i in range(label_length):
            # get the byte
            char = QNAME[index]
            # use the integer as the ascii value of the char
            url_name += chr(char)
            index += 1
        # if the next character is a null label then break
        if(QNAME[index] == 0):
            return url_name
        # otherwise add a dot
        else:
            url_name += "."
    return url_name
```

By reading byte by byte of QNAME, the first value read is assumed to be the label length. The next bytes within that length are then interpreted as ascii values. When reading the next label length if it's not a null character, a '.' character is added to the url string. The url is passed along with the request message in order to generate the header for the response. The code is as follows:

Code snippet for DNS response header

```
def getDNSHeader(request_message: bytearray, url):
    # id is first 16 bits
    request_id = request_message[0:2]
    DNS_response_header = request_id
    # hardcode the flags since these are all given beforehand
    FLAGS = 0b1000010000000000
    DNS_response_header.extend((FLAGS).to_bytes(2, 'big'))

    #QDCOUNT

    DNS_response_header.extend((0x0001).to_bytes(2, 'big'))

    #ANCOUNT
    # this changes based on how many resources

    DNS_response_header.extend((len(DNS_table[url][3])).to_bytes(2, 'big'))

    #NSCOUNT

    DNS_response_header.extend((0x0000).to_bytes(2, 'big'))

    #ARCOUNT

    DNS_response_header.extend((0x0000).to_bytes(2, 'big'))
    return DNS_response_header
```

The code is very similar to the header for the request so only the differences will be explained. QRCOUNT is set to 1 as this is a reply message. ANCOUNT is determined by looking up the DNS_table using the url from getDNSAnswers().

The header, the question from the request and the answers are then appended together to form the response message. This is sent back to the client where the client must finally parse the reply message to output the resource record(s). The code for parseData() is as follows:

Code Snippet for parseData()

```
def parseData(request: bytearray, response: bytearray) -> list:

    # parse the data
    # header is first 12 bytes
    response_header = response[0:12]
    # question is the same size as the request question
    size_of_question = len(request[12:])
    # starts from byte 12 to (12 + size of question)
    response_question = response[12: (12 + size_of_question)]
    response_answers = response[(12 + size_of_question):]

    # Find number of answers from header from ANCOUNT
    number_of_answers = response_header[6:8]
    number_of_answers = int.from_bytes(number_of_answers, 'big')

    return parseAnswers(number_of_answers, response_answers)
```

The header in the response can be quickly filtered out as only ANCOUNT from the header is required in the final print statement. The question is not needed at all and thus both ANCOUNT and the answers are passed into a function called parseAnswers.

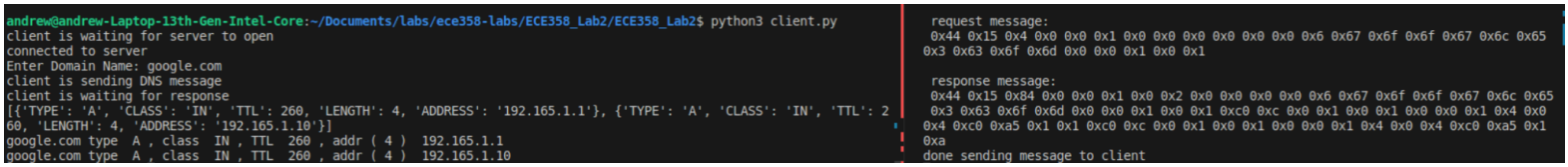
Code Snippet for parseAnswers()

```python
def parseAnswers(number_of_answers : int, response_answers : bytearray):
    results = []
    index = 0
    for i in range(number_of_answers):
        result = {}
        # Skip over the first two bytes since it's the name
        # Since it's assumed that name is 2 bytes long and constant
        index += 2

        # type is bytes 2 and 3 in the answer
        TYPE = response_answers[index: index + 2]
        index += 2

        TYPE = int.from_bytes(TYPE, 'big')
        if(TYPE == 1):
            result.update({"TYPE" : "A"})
        # class is bytes 4 and 5 in the answer
        CLASS = response_answers[index: index + 2]
        CLASS = int.from_bytes(CLASS, 'big')
        if(CLASS == 1):
            result.update({"CLASS" : "IN"})
        index += 2
        # TTL is bytes 6,7, 8 and 9
        TTL = response_answers[index: index + 4]
        TTL = int.from_bytes(TTL, 'big')
        result.update({"TTL" : TTL})
        index += 4

        # RDLENGTH is bytes 10 and 11
        RDLENGTH = response_answers[index: index + 2]
        RDLENGTH = int.from_bytes(RDLENGTH, 'big')

        result.update({"LENGTH": RDLENGTH})
        index += 2
        # RDATA is bytes 12 - 12 + RDLENGTH
        RDATA = response_answers[index: index + RDLENGTH]
        ip_string = ""
        for byte in RDATA:
            ip_string += str(byte)
            ip_string += '.'

        # remove the extra . from the string
        ip_string = ip_string[:-1]

        result.update({"ADDRESS": ip_string})
        index += RDLENGTH
        results.append(result)
    return results
```

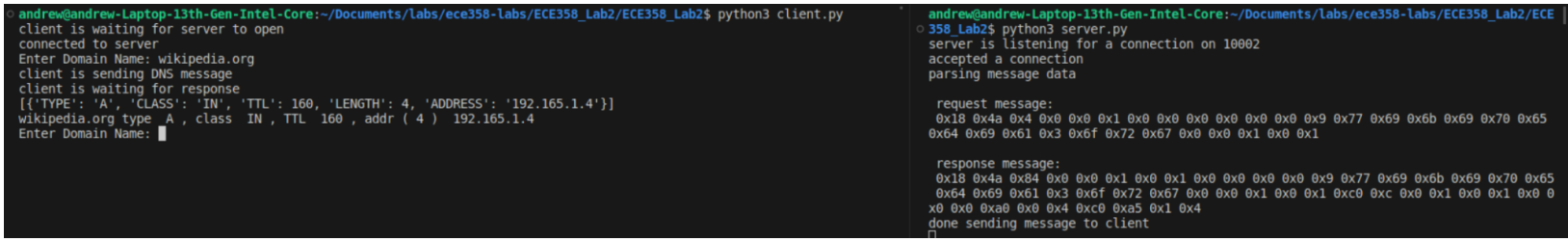
parseAnswers() takes each answer and returns a list of results. Each result is a dictionary that has 5 keys: TYPE, CLASS, TTL, LENGTH and ADDRESS. Answers are passed in as a bytearray where each answer takes up various segments of the bytearray. TYPE, CLASS, TTL and LENGTH are easy to populate since they are the [2-3], [4-5], [6-9] and [10-11] bytes of an answer respectively. The difficult part is that RDATA is of variable length and thus where each answer is within the bytearray isn't as easy as just constant offsets from each other. However, knowing this information, where one answer ends and another one begins is determined just by how large RDATA is. To determine the size of RDATA, the value of LENGTH is read. For however long LENGTH is, the next bytes in that length are taken, converted into chars and before being added to a string. Each char is separated with a '.' character where there is an extra '.' at the end which is sliced off. Finally, this string is added as the ADDRESS field of a result.

Upon parsing the answers, the results are returned and can be printed. The following are some example outputs. The output for google.com and wikipedia.org are as follows.

Terminal Screenshot of google.com



Terminal Screenshot of wikipedia.org



The wikipedia.org url is shown below for both the DNS query and DNS response.

DNS Query
request message:
0x18 0x4a 0x4 0x0 0x0 0x1 0x0 0x0 0x0 0x0 0x0 0x0 0x9 0x77 0x69 0x6b 0x69 0x70 0x65 0x64 0x69 0x61 0x3 0x6f 0x72 0x67 0x0 0x0 0x1 0x0 0x1

Color Coded DNS Query

Type	Key	Value
DNS Header	ID	0x18 0x4a
	FLAGS	0x4 0x0
	QDCOUNT	0x0 0x1
	ANCOUNT	0x0 0x0
	NSCOUNT	0x0 0x0
	ARCOUNT	0x0 0x0
QUERY	QNAME	0x9 0x77 0x69 0x6b 0x69 0x70 0x65 0x64 0x69 0x61 0x3 0x6f 0x72 0x67 0x0
	QTYPE	0x0 0x1
	QCLASS	0x0 0x1

DNS Answer

response message:
0x18 0x4a 0x84 0x0 0x0 0x1 0x0 0x1 0x0 0x0 0x0 0x0 0x9 0x77 0x69 0x6b 0x69 0x70 0x65 0x64 0x69 0x61 0x3 0x6f 0x72 0x67 0x0 0x0 0x1 0x0 0x1 0xc0 0xc 0x0 0x1 0x0 0x1 0x0 0x0 0x0 0xa0 0x0 0x4 0xc0 0xa5 0x1 0x4

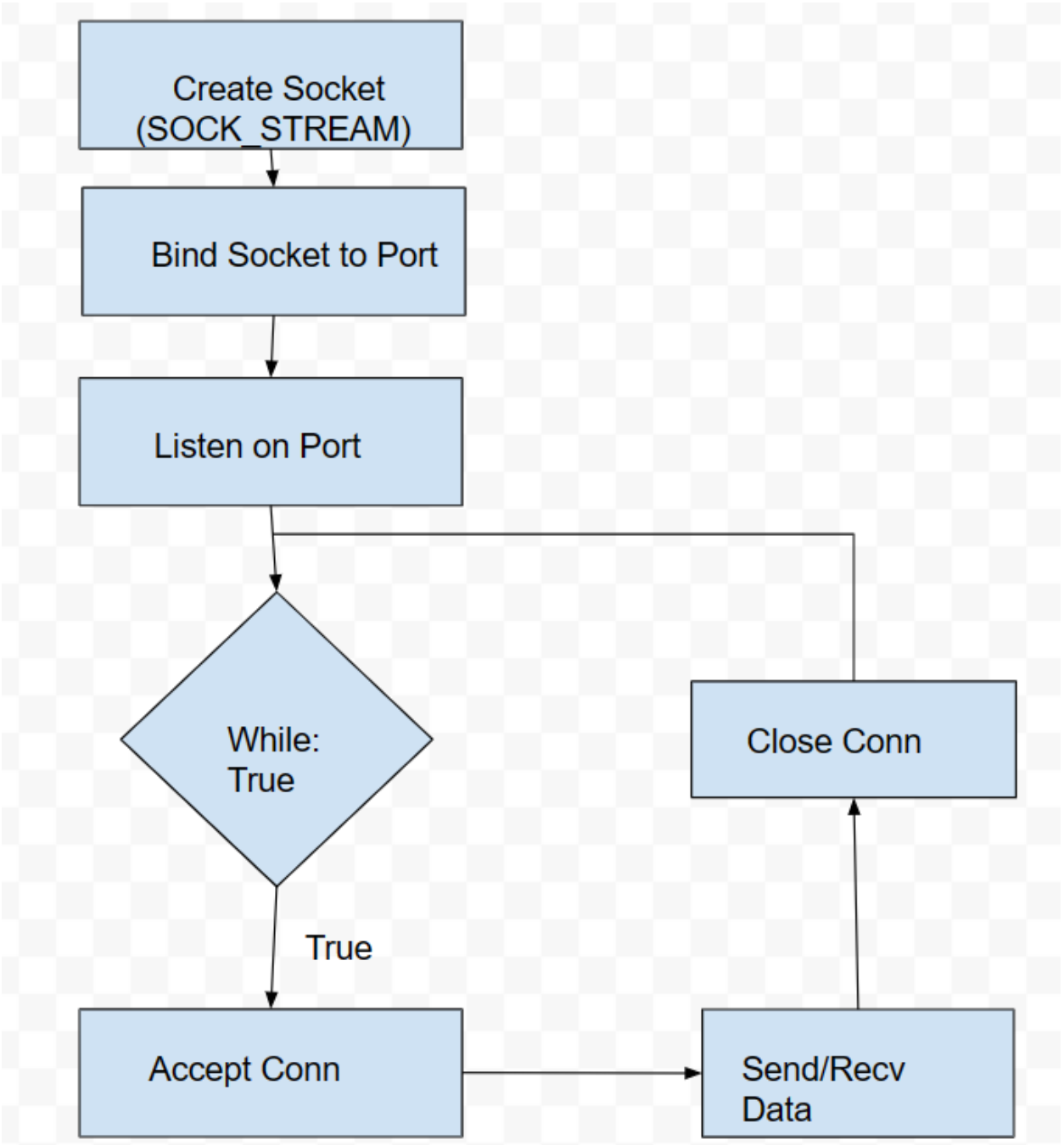
Color Coded DNS Answer

Type	Key	Value
DNS Header	ID	0x18 0x4a
	FLAGS	0x84 0x0
	QDCOUNT	0x0 0x1
	ANCOUNT	0x0 0x1
	NSCOUNT	0x0 0x0
	ARCOUNT	0x0 0x0
QUERY	QNAME	0x9 0x77 0x69 0x6b 0x69 0x70 0x65 0x64 0x69 0x61 0x3 0x6f 0x72 0x67 0x0
	QTYPE	0x0 0x1
	QCLASS	0x0 0x1
ANSWER	NAME	0xc0 0xc
	TYPE	0x0 0x1
	CLASS	0x0 0x1
	TTL	0x0 0x0 0x0 0xa0
	RDLENGTH	0x0 0x4
	RDATA	0xc0 0xa5 0x1 0x4

Task 3

1. Sockets are an abstraction to allow applications to send data over a network. It acts as an endpoint in a connection.
2. Sockets are required to prevent the developer from needing to know how the underlying transport layer is implemented. Each socket has a port number and ip address. A server and a client communicate using sockets where the server opens a socket for communication and the client connects to the server to start data transfer.
3. Datagram and Stream. Streams are synchronous and are for continuous sending of data. Stream sockets are also known as TCP Sockets. They provide all the behavior of TCP such as connection-oriented and reliable transfer of data. Datagrams are for asynchronous data transfer. Datagrams are also known as UDP sockets. It provides the connectionless behavior of UDP such as best effort delivery and high speed.
4. Socket streams can be used to make an http request. Datagrams can be used to make a DNS query.
5.

Function	**Description**
socket()	Create a new socket with a specific socket type and a specific network protocol.
bind()	Bind a socket to an address and port number. Server side only.
connect()	Connect the socket to an ip address and port. Client side only. Stream sockets only.
listen()	Listen for incoming connections on the binded address. Server side only. Stream sockets only
accept()	Accept an incoming connection and start a new connection on a new port. Server side only. Stream sockets only.
close()	Close the socket.

6. 

7. Sockets reside on the application layer but interact very closely with the transport layer.

8.

Protocol	Port #	Common Function
HTTP	80	Getting and transmitting web pages over the internet
FTP	20, 21	Transferring files between a client and a server.
IMAP4	143, 993	Accessing emails on a remote mail server.
SMTP	25, 587, 465	Send email across the internet.
Telnet	23	Remote computing.
POP3	110, 995	Download emails from a mail server.

9. DNS allows for translation of a human readable url to an ip address without having the ip address on hand. DNS stands for domain name system and will take a url and then query various databases at various locations in order to obtain the IP address of that url. The databases range from inside one's own computer to as distant as google or a large corporation's server. DNS uses UDP transport.