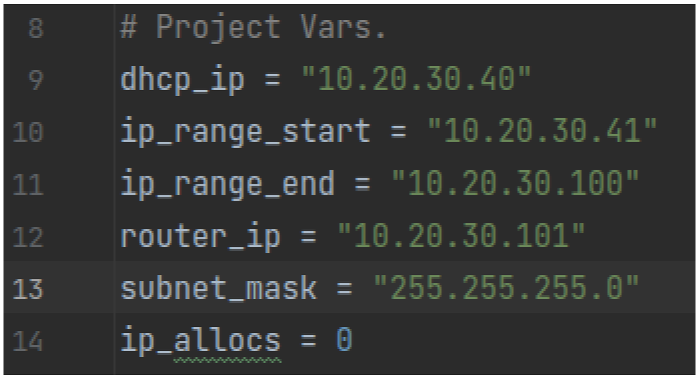
DHCP Server

DHCP server is a network server that dynamically assigns IP addresses and other devices on a network.

- A DHCP server receives and handles requests from clients on the network.
- It leases IP addresses to clients and provides them with additional configuration information such as subnet mask and default gateway.
- The DHCP server manages a pool of available IP addresses and assigns them to clients.

We’ll start with the code explanation of our DHCP Server.

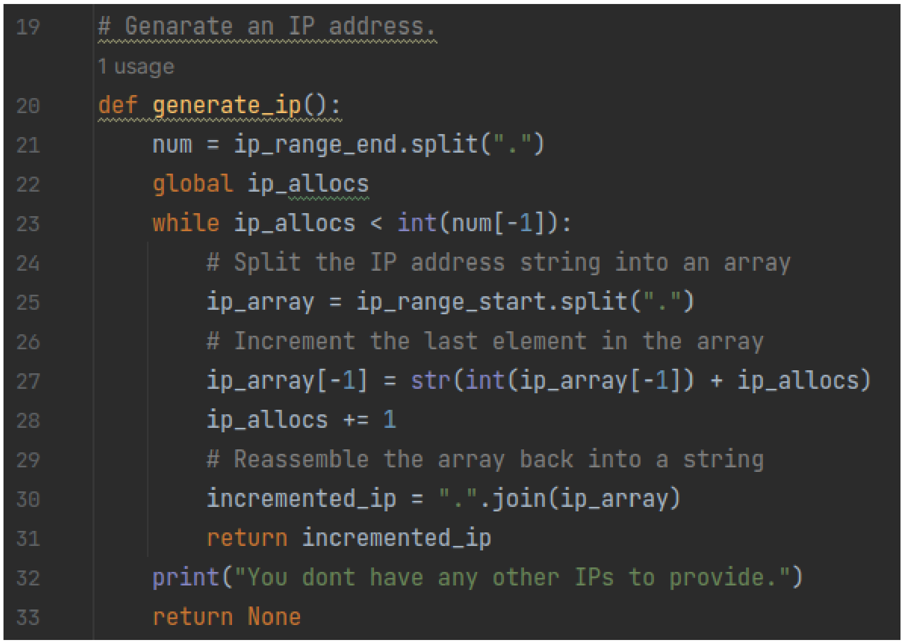
At the first few lines, all of our dynamic project vars mentioned



```
# Project Vars.
dhcp_ip = "10.20.30.40"
ip_range_start = "10.20.30.41"
ip_range_end = "10.20.30.100"
router_ip = "10.20.30.101"
subnet_mask = "255.255.255.0"
ip_allocs = 0
```

Starting from the actual DHCP Server IP and followed by the range of IP addresses that the DHCP will offer, Subnet Mask, Router IP and at the end that will be a counter that will help to keep track of our dispatched IP’s.

Generate IP Function



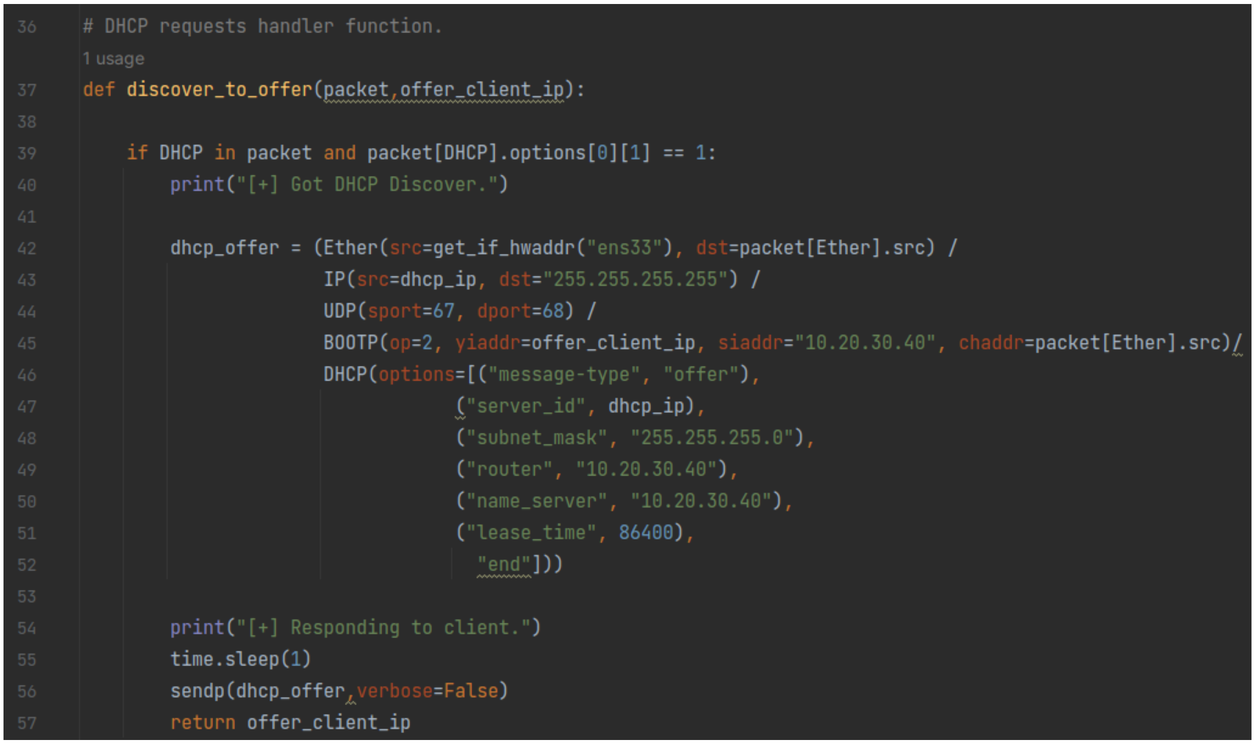
```python
# Genarate an IP address.
def generate_ip():
    num = ip_range_end.split(".")
    global ip_allocs
    while ip_allocs < int(num[-1]):
        # Split the IP address string into an array
        ip_array = ip_range_start.split(".")
        # Increment the last element in the array
        ip_array[-1] = str(int(ip_array[-1]) + ip_allocs)
        ip_allocs += 1
        # Reassemble the array back into a string
        incremented_ip = ".".join(ip_array)
        return incremented_ip
    print("You dont have any other IPs to provide.")
    return None
```

As mentioned above, Using the three vars from the top section:

The addresses will be generated by the “hard coded” range, from ip_range_start to ip_range_end.   The Function is a pretty simple manipulation of those ip_range strings,

We split the ip_range_end string by a “.” To get the number that will be the upper boundary of our IP addresses. Then returning the next IP in the sequence and checking that we are not overflowing out of range.

DHCP Requests Handler:



```
36  # DHCP requests handler function.
    1 usage
37  def discover_to_offer(packet,offer_client_ip):
38
39      if DHCP in packet and packet[DHCP].options[0][1] == 1:
40          print("[+] Got DHCP Discover.")
41
42          dhcp_offer = (Ether(src=get_if_hwaddr("ens33"), dst=packet[Ether].src) /
43                        IP(src=dhcp_ip, dst="255.255.255.255") /
44                        UDP(sport=67, dport=68) /
45                        BOOTP(op=2, yiaddr=offer_client_ip, siaddr="10.20.30.40", chaddr=packet[Ether].src)/
46                        DHCP(options=[("message-type", "offer"),
47                                      ("server_id", dhcp_ip),
48                                      ("subnet_mask", "255.255.255.0"),
49                                      ("router", "10.20.30.40"),
50                                      ("name_server", "10.20.30.40"),
51                                      ("lease_time", 86400),
52                                        "end"]))
53
54          print("[+] Responding to client.")
55          time.sleep(1)
56          sendp(dhcp_offer,verbose=False)
57          return offer_client_ip
```

This function has two vars that's being passed as an argument.

Paket stands for the actual packet that should be handled and the Client IP address.

Line 39 checks if the packet contains a DHCP message and if the DHCP message type is 'Discover'.

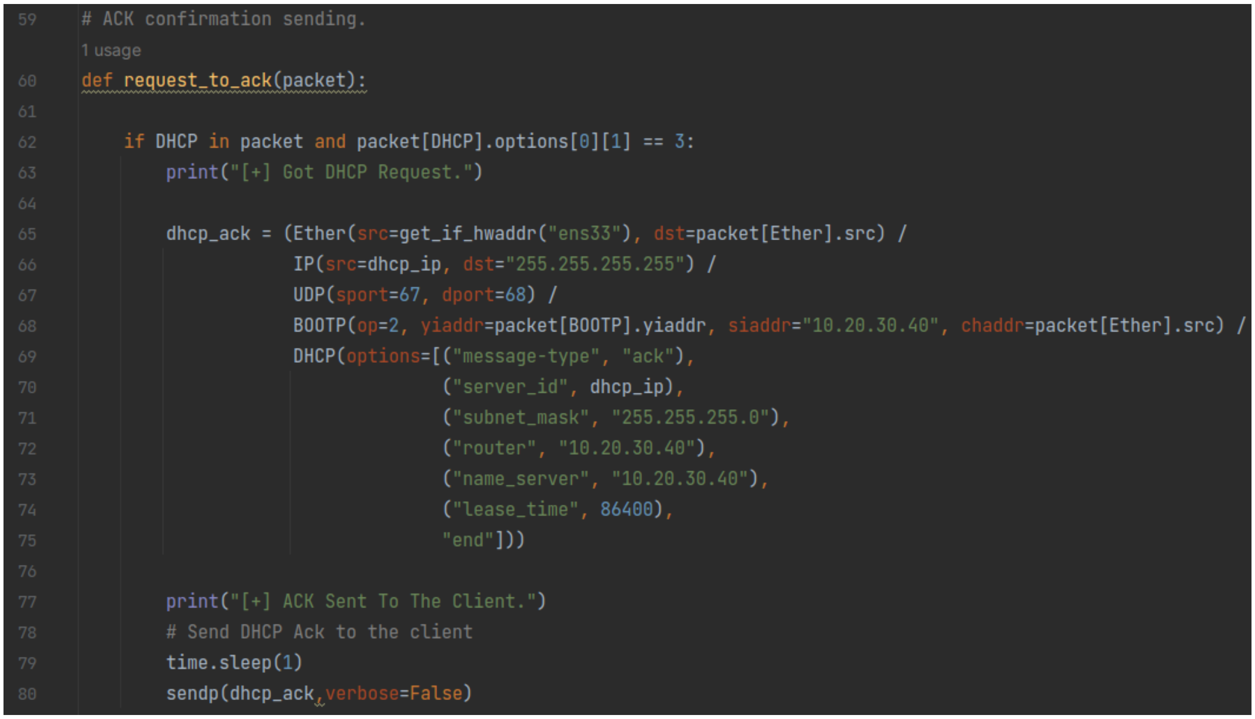
Line 42 to 52 Building the DHCP offer packet including HEADER (Ethernet, IP, UDP, BOOTP and DHCP) with all the necessary data.

* Line 55 is a waiting command, we used it to avoid the packet stream that happens before the function call.

Line 56 Scapy built-in function that sends the DHCP the offer that was made just above.

Line 57 Returning the actual IP address that was offered.

ACK Confirmation Sending:



```
59  # ACK confirmation sending.
    1 usage
60  def request_to_ack(packet):
61
62      if DHCP in packet and packet[DHCP].options[0][1] == 3:
63          print("[+] Got DHCP Request.")
64
65          dhcp_ack = (Ether(src=get_if_hwaddr("ens33"), dst=packet[Ether].src) /
66                      IP(src=dhcp_ip, dst="255.255.255.255") /
67                      UDP(sport=67, dport=68) /
68                      BOOTP(op=2, yiaddr=packet[BOOTP].yiaddr, siaddr="10.20.30.40", chaddr=packet[Ether].src) /
69                      DHCP(options=[("message-type", "ack"),
70                                    ("server_id", dhcp_ip),
71                                    ("subnet_mask", "255.255.255.0"),
72                                    ("router", "10.20.30.40"),
73                                    ("name_server", "10.20.30.40"),
74                                    ("lease_time", 86400),
75                                    "end"]))
76
77          print("[+] ACK Sent To The Client.")
78          # Send DHCP Ack to the client
79          time.sleep(1)
80          sendp(dhcp_ack,verbose=False)
```

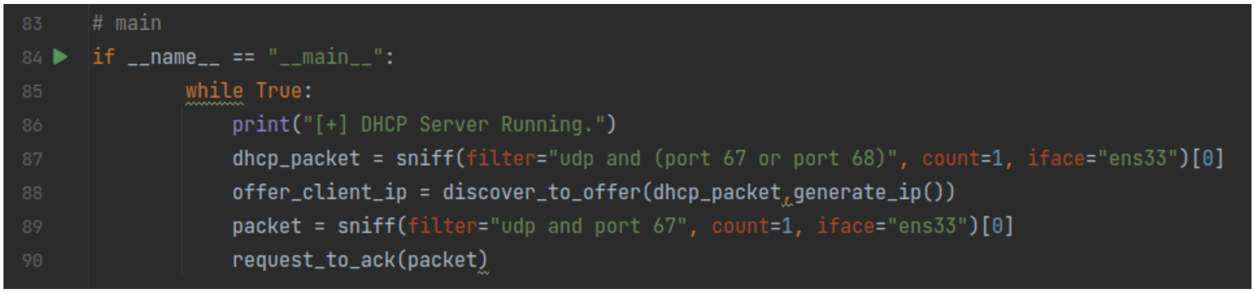
Line 62 checks if the packet contains a DHCP message and if the DHCP message type is 'Request'.

Line 65 to 75 Building the DHCP ACK packet including HEADER (Ethernet, IP, UDP, BOOTP and DHCP) with all the necessary data.

* Line 79 is a waiting command, we used it to avoid the packet stream that happens before the function call.

Line 80 Scapy built-in function that sends the DHCP ACK that was made just above.

Main function:



```
83   # main
84   if __name__ == "__main__":
85           while True:
86               print("[+] DHCP Server Running.")
87               dhcp_packet = sniff(filter="udp and (port 67 or port 68)", count=1, iface="ens33")[0]
88               offer_client_ip = discover_to_offer(dhcp_packet,generate_ip())
89               packet = sniff(filter="udp and port 67", count=1, iface="ens33")[0]
90               request_to_ack(packet)
```

Line 85 starts with a while loop that contains DHCP packet sniffing using the ‘sniff’ tool from Scapy library.

Followed by the ‘offer_client_ip’ var that captured the returned value from the discover_to_offer function explained above.

Var ‘packet’ captures the client packet using the same ‘sniff’ by Scapy for to be analyzed.

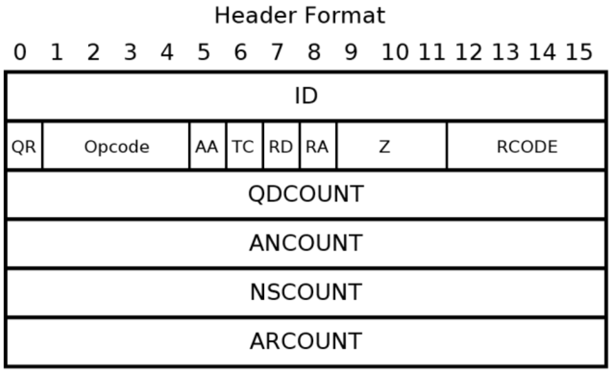
Line 90 calling the ‘request_to_ack’ explained above as well with the packet var to be analyzed.

DNS Server

A DNS server is a network server that translates domain into IP addresses.

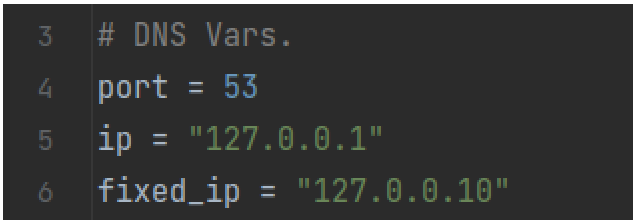
- A DNS server receives requests from client devices looking to resolve domain names into IP addresses.
- It holds a database of domain names and IP addresses and returns the requested IP address to the client.

To build the DNS packet as a response to the client we will walk through each flag.



Code explanation:

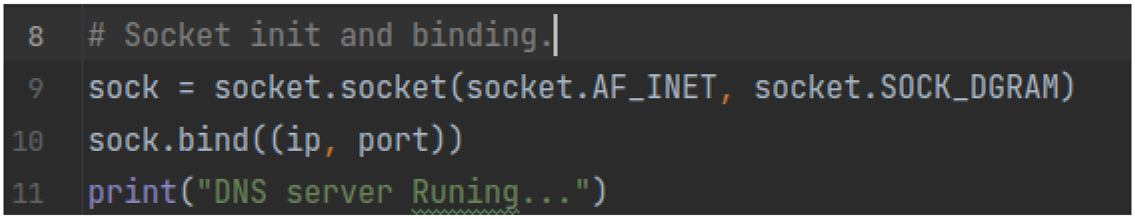
The DNS server starts with:



```
# DNS Vars.
port = 53
ip = "127.0.0.1"
fixed_ip = "127.0.0.10"
```

A few vars that the only one that's should be explained is the 'fixed', this var holds the proxy server address.

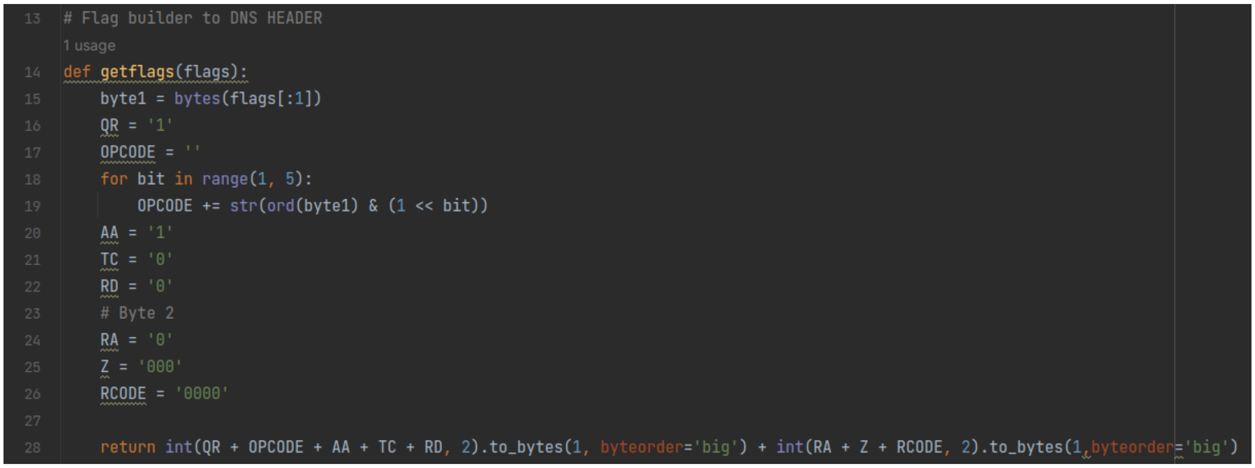
Followed by:



```
# Socket init and binding.
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
sock.bind((ip, port))
print("DNS server Runing...")
```

The DNS Server socket initialization and binding.

Flag builder to DNS HEADER:



```
13  # Flag builder to DNS HEADER
    1 usage
14  def getflags(flags):
15      byte1 = bytes(flags[:1])
16      QR = '1'
17      OPCODE = ''
18      for bit in range(1, 5):
19          OPCODE += str(ord(byte1) & (1 << bit))
20      AA = '1'
21      TC = '0'
22      RD = '0'
23      # Byte 2
24      RA = '0'
25      Z = '000'
26      RCODE = '0000'
27
28      return int(QR + OPCODE + AA + TC + RD, 2).to_bytes(1, byteorder='big') + int(RA + Z + RCODE, 2).to_bytes(1,byteorder='big')
```

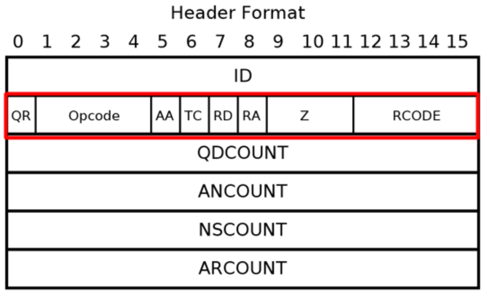
In this function we are about to

build the first byte.

(QR, Optcode, AA, TC, RD)

Second byte.

(RA, Z, RCODE)



Header Format

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

ID

QR | Opcode | AA | TC | RD | RA | Z | RCODE

QDCOUNT

ANCOUNT

NSCOUNT

ARCOUNT

Line 15 Creates a bytes object ‘byte1’ that representing the first byte.

Flags:

QR is set to '1', indicating that this is a response message.

OPCODE is an empty string and will be set later based on the value of bits 2-5 in the first byte of the flags byte string.

AA, TC, and RD are set to '1', '0', and '0', respectively, indicating that this is an authoritative response.

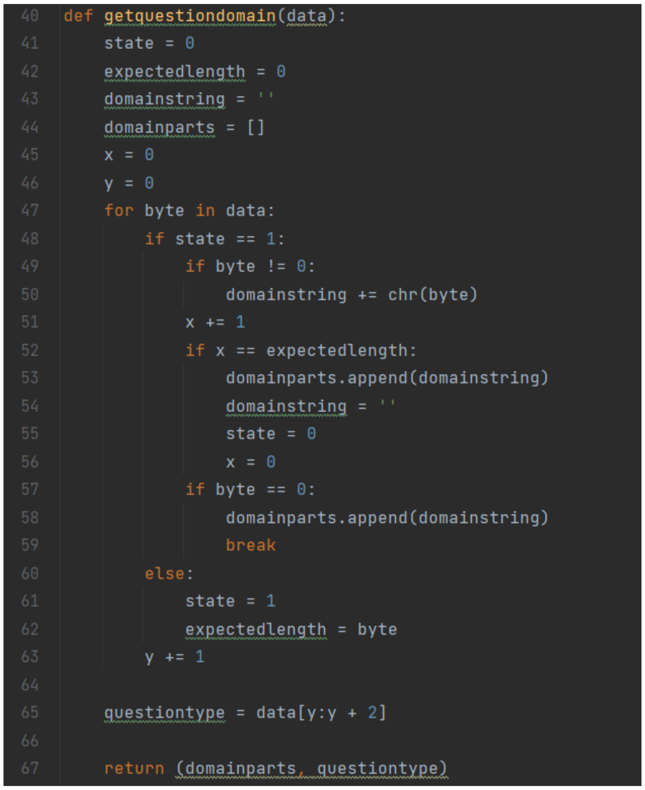
RA is set to '0' indicating that recursion is not available.

Z is 3 bits reserved as an option but the default is ‘000’

RCODE is set to '0000' indicates a valid response.

Line 28 Returning the constructed value in binary (Big Endian) include the first and second byte data.

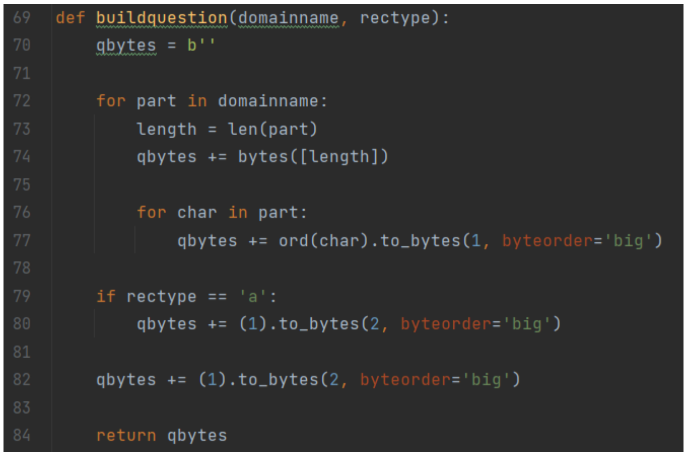
Flag builder to DNS HEADER:



```python
def getquestiondomain(data):
    state = 0
    expectedlength = 0
    domainstring = ''
    domainparts = []
    x = 0
    y = 0
    for byte in data:
        if state == 1:
            if byte != 0:
                domainstring += chr(byte)
            x += 1
            if x == expectedlength:
                domainparts.append(domainstring)
                domainstring = ''
                state = 0
                x = 0
            if byte == 0:
                domainparts.append(domainstring)
                break
        else:
            state = 1
            expectedlength = byte
        y += 1

    questiontype = data[y:y + 2]

    return (domainparts, questiontype)
```

The main use of the function is to get the data that was submitted to it by the argument and determine if the data is belongs to A type or AAAA type queries.

Build Question:

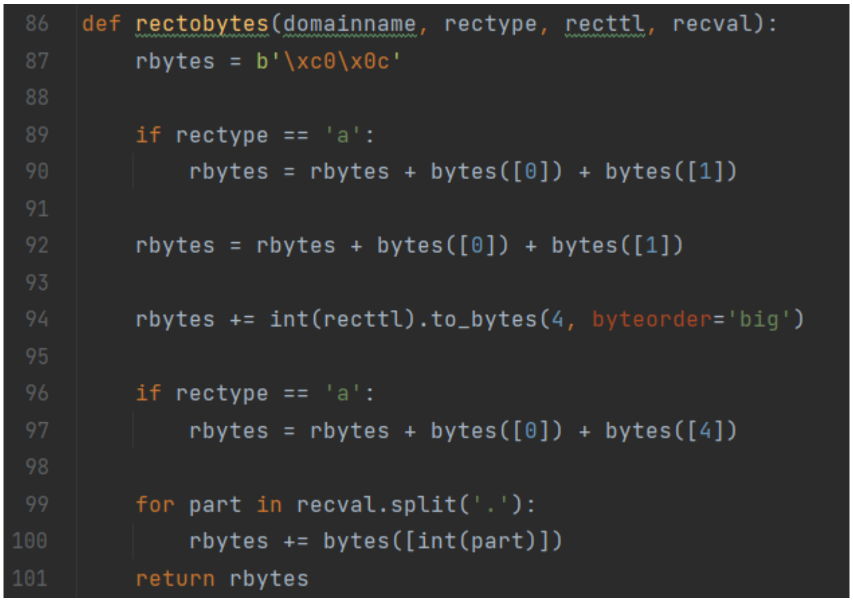


```python
def buildquestion(domainname, rectype):
    qbytes = b''

    for part in domainname:
        length = len(part)
        qbytes += bytes([length])

        for char in part:
            qbytes += ord(char).to_bytes(1, byteorder='big')

    if rectype == 'a':
        qbytes += (1).to_bytes(2, byteorder='big')

    qbytes += (1).to_bytes(2, byteorder='big')

    return qbytes
```

Getting as an argument the name of the Domain and the type (A or AAAA).

This function will build the query part of the response packet that was sent by the DNS. (We made this function to make the user be able to see know the question for every answer he got)

Rec to Bytes:



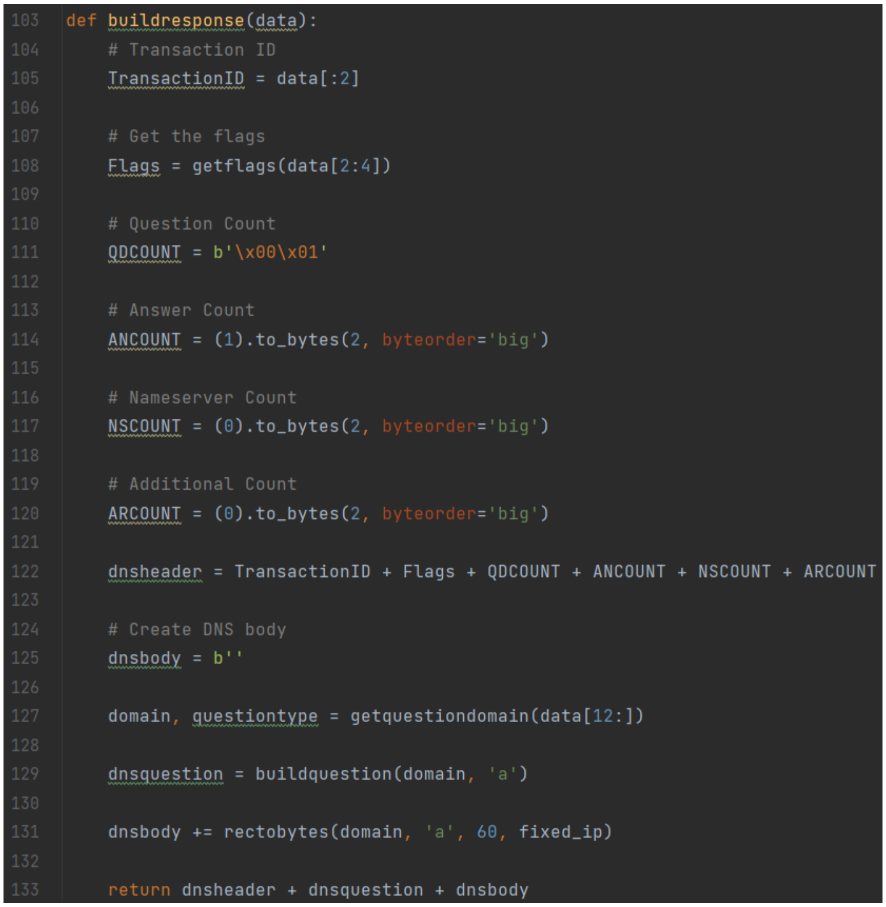
```python
def rectobytes(domainname, rectype, recttl, recval):
    rbytes = b'\xc0\x0c'

    if rectype == 'a':
        rbytes = rbytes + bytes([0]) + bytes([1])

    rbytes = rbytes + bytes([0]) + bytes([1])

    rbytes += int(recttl).to_bytes(4, byteorder='big')

    if rectype == 'a':
        rbytes = rbytes + bytes([0]) + bytes([4])

    for part in recval.split('.'):
        rbytes += bytes([int(part)])
    return rbytes
```

This function gest 4 values at the argument,

domain – Domain name, rectype – A or AAA type query, recttl – TTL, recval – IP value for type A.

Building all the bytes that corresponding to this section.

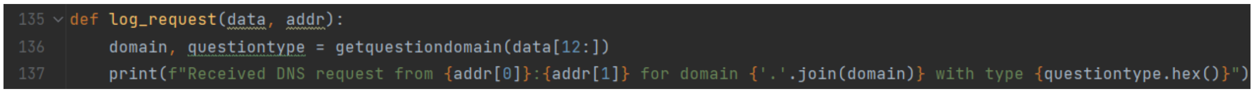
Build the response:



```python
def buildresponse(data):
    # Transaction ID
    TransactionID = data[:2]

    # Get the flags
    Flags = getflags(data[2:4])

    # Question Count
    QDCOUNT = b'\x00\x01'

    # Answer Count
    ANCOUNT = (1).to_bytes(2, byteorder='big')

    # Nameserver Count
    NSCOUNT = (0).to_bytes(2, byteorder='big')

    # Additional Count
    ARCOUNT = (0).to_bytes(2, byteorder='big')

    dnsheader = TransactionID + Flags + QDCOUNT + ANCOUNT + NSCOUNT + ARCOUNT

    # Create DNS body
    dnsbody = b''

    domain, questiontype = getquestiondomain(data[12:])

    dnsquestion = buildquestion(domain, 'a')

    dnsbody += rectobytes(domain, 'a', 60, fixed_ip)

    return dnsheader + dnsquestion + dnsbody
```

This function will take the data that was passed in the argument and will build the DNS response from it.

Log Request:



```python
def log_request(data, addr):
    domain, questiontype = getquestiondomain(data[12:])
    print(f"Received DNS request from {addr[0]}:{addr[1]} for domain {'.'.join(domain)} with type {questiontype.hex()}")
```

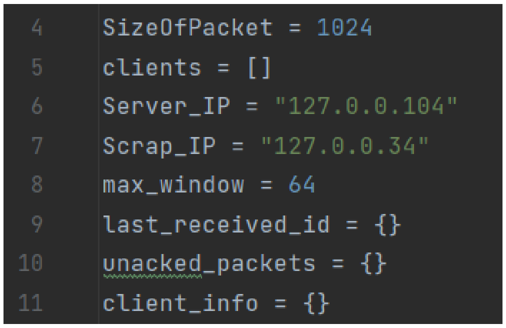
Prints the Client IP address, the Domain that the client seek to join and type of the query.

RUDP/Proxy Server

The Proxy/RUDP Server main cause is to be the middleman of the connection between the Download Server and the Client.

Will be communicating to the Client by with RUDP (UDP + ACK + CC + FC) based connection and communicating with the Scrap Program with TCP connection based.

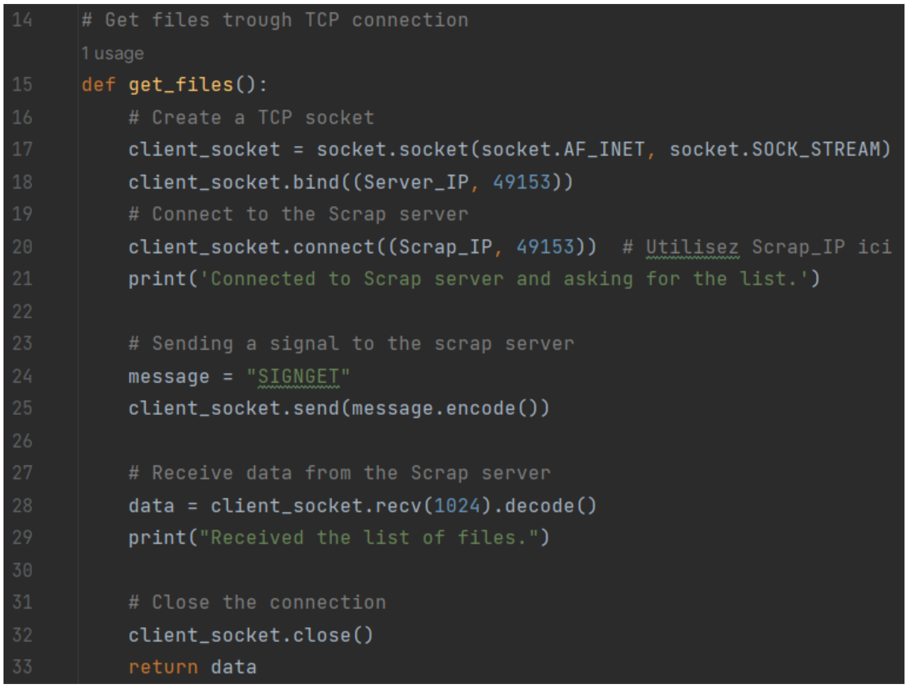
Code explanation:



```python
SizeOfPacket = 1024
clients = []
Server_IP = "127.0.0.104"
Scrap_IP = "127.0.0.34"
max_window = 64
last_received_id = {}
unacked_packets = {}
client_info = {}
```

The main and global variables of the Proxy.

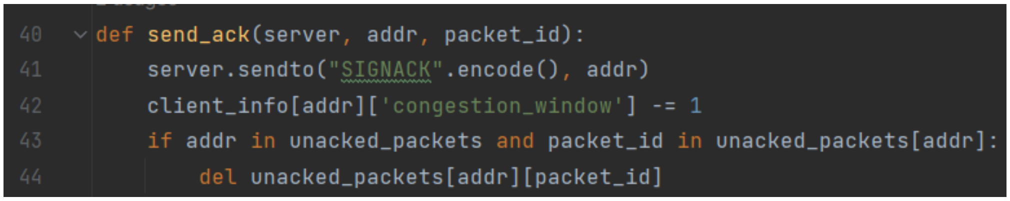
Downloadable files list maker:



```python
# Get files trough TCP connection
def get_files():
    # Create a TCP socket
    client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    client_socket.bind((Server_IP, 49153))
    # Connect to the Scrap server
    client_socket.connect((Scrap_IP, 49153))  # Utilisez Scrap_IP ici
    print('Connected to Scrap server and asking for the list.')

    # Sending a signal to the scrap server
    message = "SIGNGET"
    client_socket.send(message.encode())

    # Receive data from the Scrap server
    data = client_socket.recv(1024).decode()
    print("Received the list of files.")

    # Close the connection
    client_socket.close()
    return data
```

This function will establish a TCP connection with the Scrap Program to get the list of files.

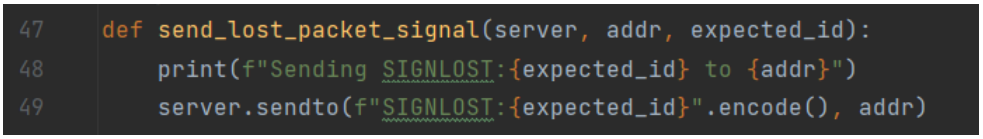
Send ACK:



```python
def send_ack(server, addr, packet_id):
    server.sendto("SIGNACK".encode(), addr)
    client_info[addr]['congestion_window'] -= 1
    if addr in unacked_packets and packet_id in unacked_packets[addr]:
        del unacked_packets[addr][packet_id]
```

Will send an ACK when there was a request to open or close the connection.

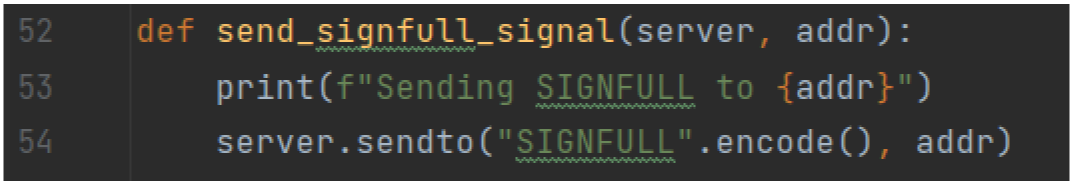
Send Lost Packet Signal:



```python
def send_lost_packet_signal(server, addr, expected_id):
    print(f"Sending SIGNLOST:{expected_id} to {addr}")
    server.sendto(f"SIGNLOST:{expected_id}".encode(), addr)
```

Will send a Signal to the client "SIGNLOST" to notify that was a packet loss. (The packet include the ID and type)

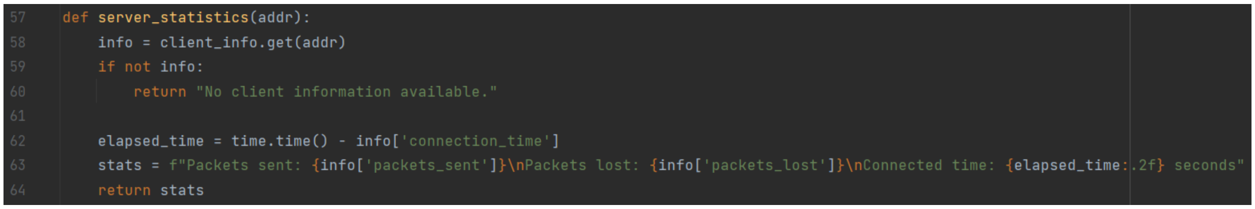
Send Sign Full Signal:



```python
def send_signfull_signal(server, addr):
    print(f"Sending SIGNFULL to {addr}")
    server.sendto("SIGNFULL".encode(), addr)
```

Sending a signal "SIGNFULL" to notify the client that the window is full and to delay the further sending packets.

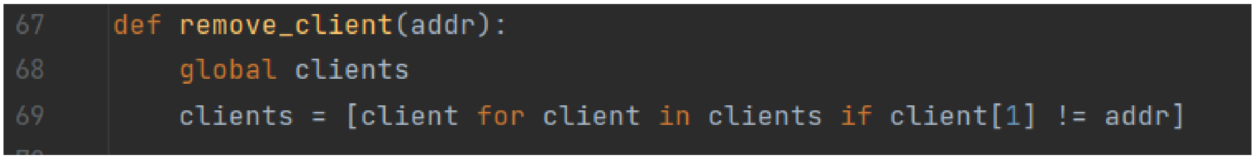
Server Statistics:



```python
def server_statistics(addr):
    info = client_info.get(addr)
    if not info:
        return "No client information available."

    elapsed_time = time.time() - info['connection_time']
    stats = f"Packets sent: {info['packets_sent']}\nPackets lost: {info['packets_lost']}\nConnected time: {elapsed_time:.2f} seconds"
    return stats
```

Building the message when the signal "SIGNSTAT" is sent by the client.

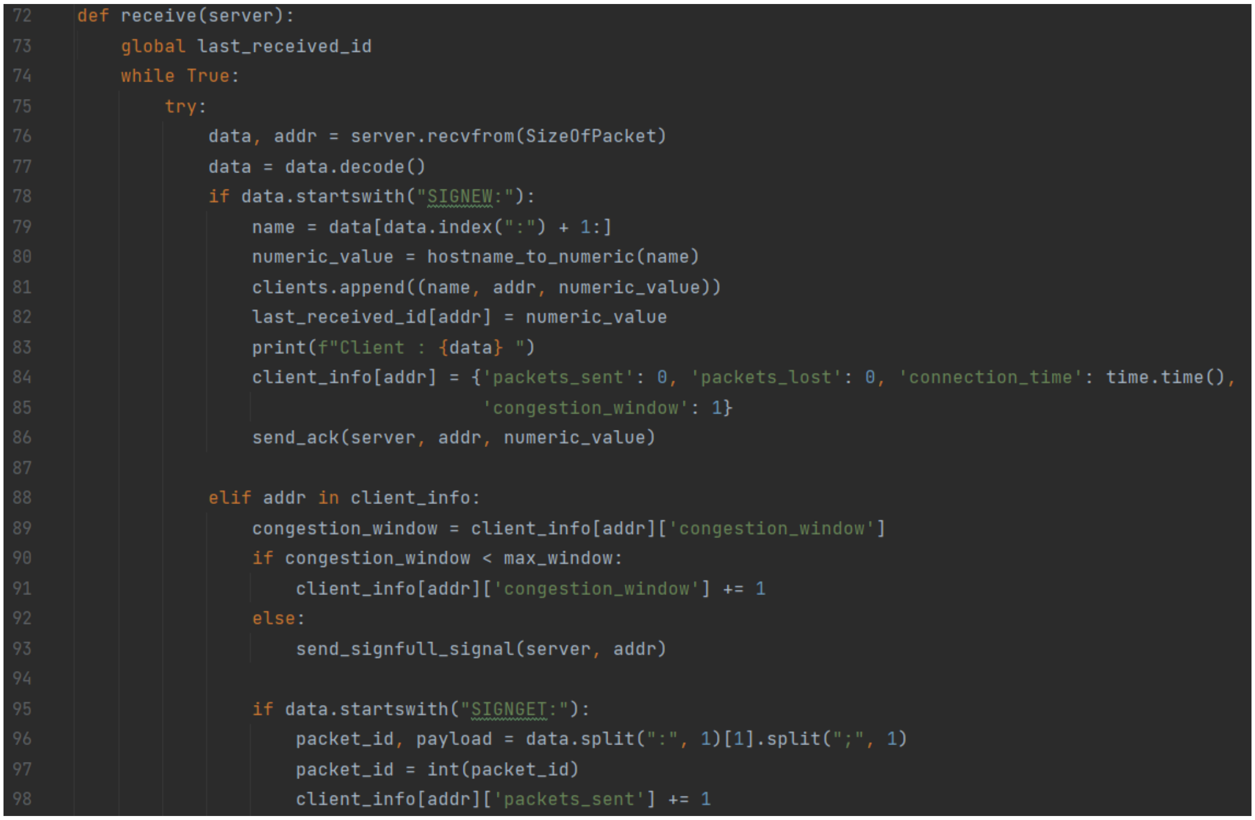
Remove Client:



```
def remove_client(addr):
    global clients
    clients = [client for client in clients if client[1] != addr]
```

Removes the client form the array by there IP and PORT.

Main RUDP Server Function - Receive:



```
def receive(server):
    global last_received_id
    while True:
        try:
            data, addr = server.recvfrom(SizeOfPacket)
            data = data.decode()
            if data.startswith("SIGNEW:"):
                name = data[data.index(":") + 1:]
                numeric_value = hostname_to_numeric(name)
                clients.append((name, addr, numeric_value))
                last_received_id[addr] = numeric_value
                print(f"Client : {data} ")
                client_info[addr] = {'packets_sent': 0, 'packets_lost': 0, 'connection_time': time.time(),
                                     'congestion_window': 1}
                send_ack(server, addr, numeric_value)

            elif addr in client_info:
                congestion_window = client_info[addr]['congestion_window']
                if congestion_window < max_window:
                    client_info[addr]['congestion_window'] += 1
                else:
                    send_signfull_signal(server, addr)

                if data.startswith("SIGNGET:"):
                    packet_id, payload = data.split(":", 1)[1].split(";", 1)
                    packet_id = int(packet_id)
                    client_info[addr]['packets_sent'] += 1
```

The first part defines the main methods used for sniffing every request sent by the Client.

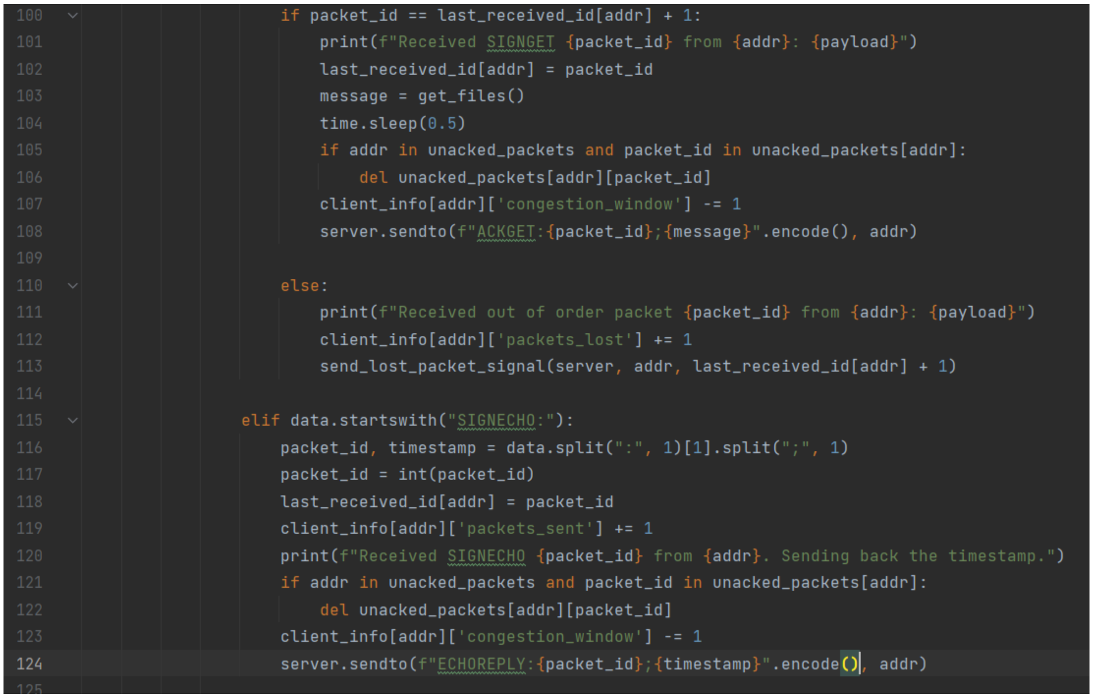
Checks for a known keyword to know what type of request the client sent.

“SIGNEW” – will create a new connection and will be operating on a request to close the connection as well,

Will save all the information about the client in the client info list.

Line 88 to 93 - Will check the window to verify that there is enough space.

“SIGNGET” – will make the request for the list of downloadable files.



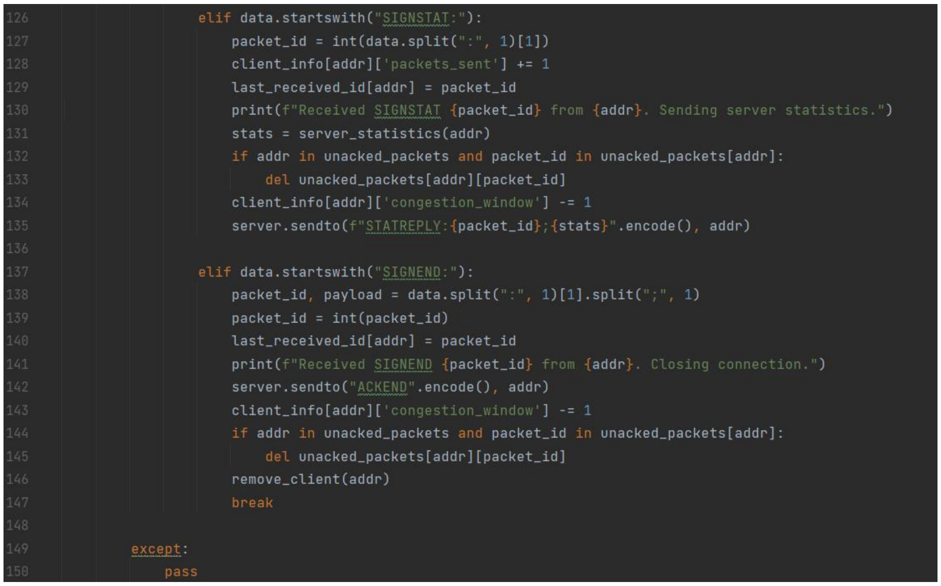
```
                if packet_id == last_received_id[addr] + 1:
                    print(f"Received SIGNGET {packet_id} from {addr}: {payload}")
                    last_received_id[addr] = packet_id
                    message = get_files()
                    time.sleep(0.5)
                    if addr in unacked_packets and packet_id in unacked_packets[addr]:
                        del unacked_packets[addr][packet_id]
                    client_info[addr]['congestion_window'] -= 1
                    server.sendto(f"ACKGET:{packet_id};{message}".encode(), addr)

                else:
                    print(f"Received out of order packet {packet_id} from {addr}: {payload}")
                    client_info[addr]['packets_lost'] += 1
                    send_lost_packet_signal(server, addr, last_received_id[addr] + 1)

            elif data.startswith("SIGNECHO:"):
                packet_id, timestamp = data.split(":", 1)[1].split(";", 1)
                packet_id = int(packet_id)
                last_received_id[addr] = packet_id
                client_info[addr]['packets_sent'] += 1
                print(f"Received SIGNECHO {packet_id} from {addr}. Sending back the timestamp.")
                if addr in unacked_packets and packet_id in unacked_packets[addr]:
                    del unacked_packets[addr][packet_id]
                client_info[addr]['congestion_window'] -= 1
                server.sendto(f"ECHOREPLY:{packet_id};{timestamp}".encode(), addr)
```

Firstly validates the packet sequence of the packet ID, then will make the request for the downloadable file list.

Sends an “ACKGET” with the file list and updates the Server’s window, otherwise will send a signal that there is a packet loss.

“SIGNECHO” – Will show the statistics gathered from performing PING between the Client and the Proxy.

The Proxy Server will send to the Client the packet that include the timestamps.



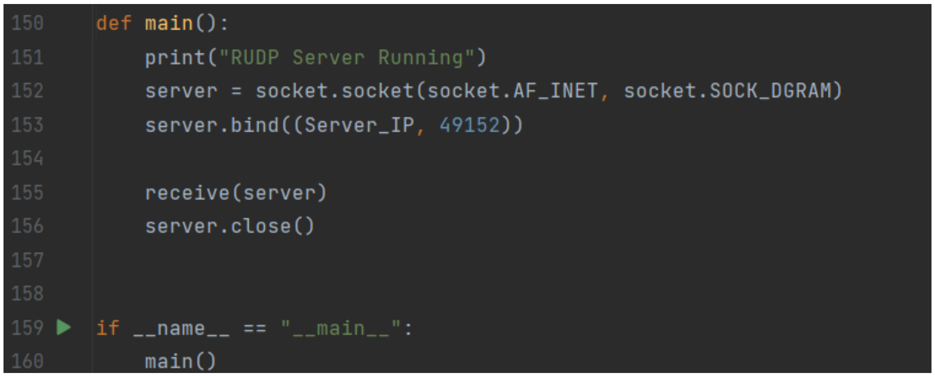
```python
                    elif data.startswith("SIGNSTAT:"):
                        packet_id = int(data.split(":", 1)[1])
                        client_info[addr]['packets_sent'] += 1
                        last_received_id[addr] = packet_id
                        print(f"Received SIGNSTAT {packet_id} from {addr}. Sending server statistics.")
                        stats = server_statistics(addr)
                        if addr in unacked_packets and packet_id in unacked_packets[addr]:
                            del unacked_packets[addr][packet_id]
                        client_info[addr]['congestion_window'] -= 1
                        server.sendto(f"STATREPLY:{packet_id};{stats}".encode(), addr)

                    elif data.startswith("SIGNEND:"):
                        packet_id, payload = data.split(":", 1)[1].split(";", 1)
                        packet_id = int(packet_id)
                        last_received_id[addr] = packet_id
                        print(f"Received SIGNEND {packet_id} from {addr}. Closing connection.")
                        server.sendto("ACKEND".encode(), addr)
                        client_info[addr]['congestion_window'] -= 1
                        if addr in unacked_packets and packet_id in unacked_packets[addr]:
                            del unacked_packets[addr][packet_id]
                        remove_client(addr)
                        break

            except:
                pass
```

“SIGNSTAT” – Will print to the screen all the information about the number of packets that the Client sent, number of lost packets and the time of live connection between client and Proxy/RUDP Server.

The Proxy Server will send a packet with the SIGNSTAT reply and the information about the statistics.

“SIGNEND” – Will notify the Proxy that the client would like to end the connection.

Main function:



```python
def main():
    print("RUDP Server Running")
    server = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    server.bind((Server_IP, 49152))

    receive(server)
    server.close()


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

Initializing the socket and binding it.

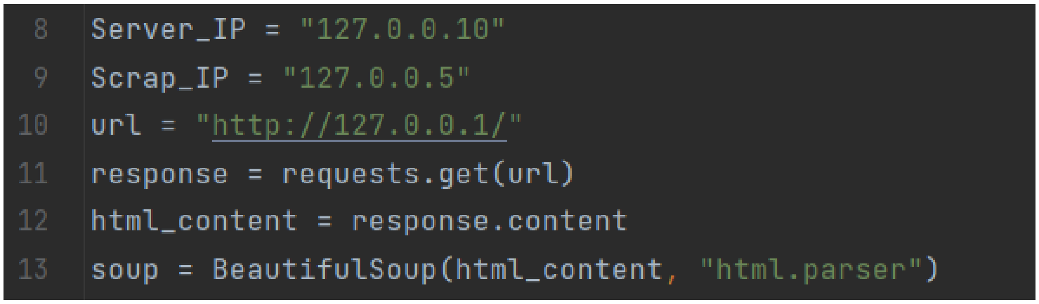
Calling the Receive function with the socket information.

SCRAP Program

The scrap server is used as a program to help for gathering information form the HTTP server that works as a website that shows all the files available for download.

Plus downloading the actuals files into the project folder using “Wget” - a built-in Linux program.

Code explanation:



```
Server_IP = "127.0.0.10"
Scrap_IP = "127.0.0.5"
url = "http://127.0.0.1/"
response = requests.get(url)
html_content = response.content
soup = BeautifulSoup(html_content, "html.parser")
```

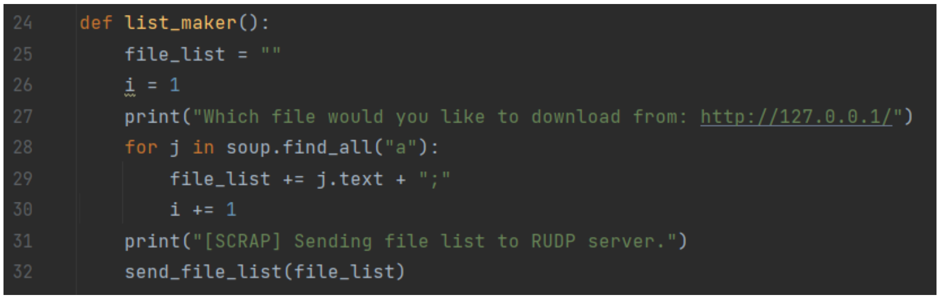
Lines 8 to 13 are the project variables holding the Proxy server IP address presented as 'Server_IP' and the actual IP address for the Scrap program itself, URL of the HTTP server.

response = requests.get(url) - Sends an HTTP request to the URL in ask for the page content, HEADER and status code.

html_content = response.content – Extracts the source code of the HTML page that the HTTP server holds as it's index page.

soup = BeautifulSoup(html_content, "html.parser") creates an BS object from the HTML content to to able to search the page content for the wanted tags, in our case 'a' tags.

Downloadable files list maker:



```
24  def list_maker():
25      file_list = ""
26      i = 1
27      print("Which file would you like to download from: http://127.0.0.1/")
28      for j in soup.find_all("a"):
29          file_list += j.text + ";"
30          i += 1
31      print("[SCRAP] Sending file list to RUDP server.")
32      send_file_list(file_list)
```

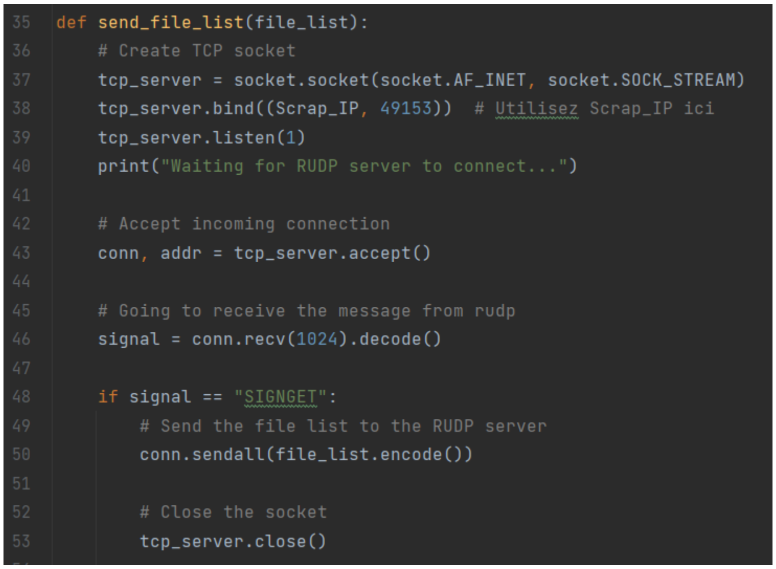
Using the great library of 'bs4' trough 'BeautifulSoup'

Line 16 to 19 - The actual file scrapper that was implemented by a for loop to search the index page of the HTTP server for an 'a' TAGS, each a tag holds a name of file and the path for it.

Each iteration we are adding the file names to a string.

Line 20 - "Sending" the constructed string to the next function that will be explained below.

Downloadable files list maker:



```
35 def send_file_list(file_list):
36     # Create TCP socket
37     tcp_server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
38     tcp_server.bind((Scrap_IP, 49153))  # Utilisez Scrap_IP ici
39     tcp_server.listen(1)
40     print("Waiting for RUDP server to connect...")
41 
42     # Accept incoming connection
43     conn, addr = tcp_server.accept()
44 
45     # Going to receive the message from rudp
46     signal = conn.recv(1024).decode()
47 
48     if signal == "SIGNGET":
49         # Send the file list to the RUDP server
50         conn.sendall(file_list.encode())
51 
52         # Close the socket
53         tcp_server.close()
```

Line 37 to 39 – Establishing connection trough declaring a socket binding and listening for calls that further on We'll be sending the information that holds the files available to download to the client.

Line 46, 48 to 53 – We are working our way trough few keyworks that each of them will preform a task,

In this case "SIGNGET" will be calling the function to send to the Proxy server the list of files that at the end the user will be able to download.

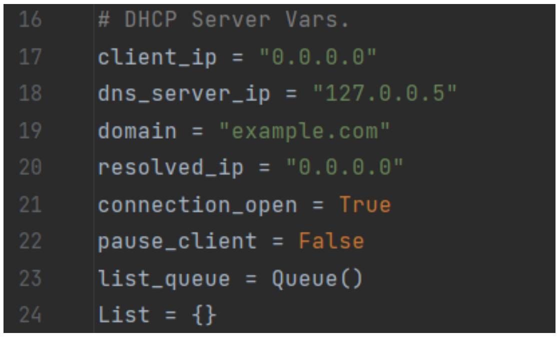
CLIENT

The client preforming a few tasks one after another,

Firstly will send a DHCP Request to get IP address from the DHCP Server (That's already on and waiting for these type of requests), in the same time making a call for the DNS server to get the list/database of the connected devices to the network that the DNS holds.

Right after It establishing a connection with our Proxy server using our RUDP socket that was created from scratch in order to get information on the files that's are available to download.

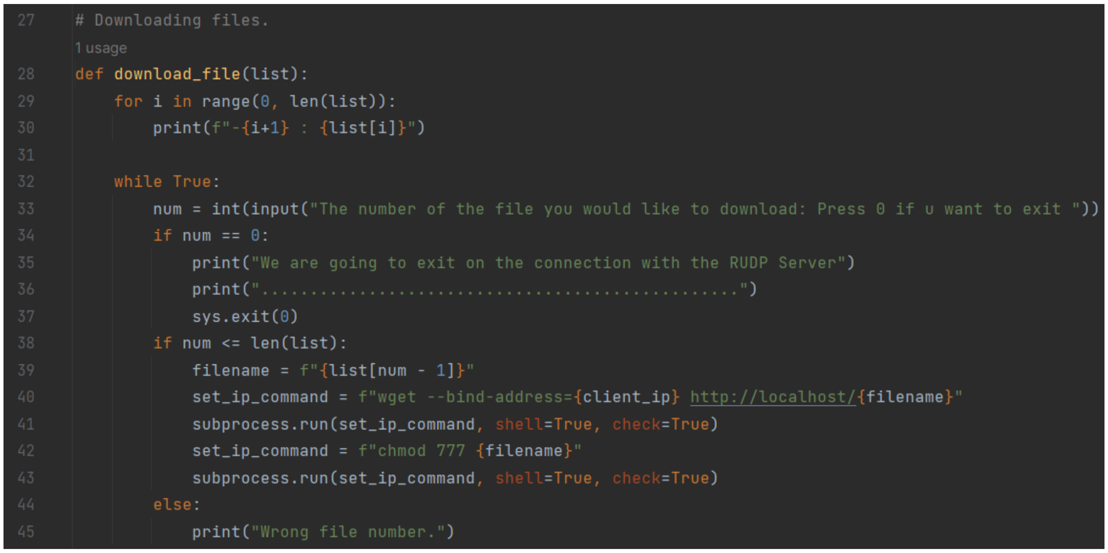
Code explanation:



```
16    # DHCP Server Vars.
17    client_ip = "0.0.0.0"
18    dns_server_ip = "127.0.0.5"
19    domain = "example.com"
20    resolved_ip = "0.0.0.0"
21    connection_open = True
22    pause_client = False
23    list_queue = Queue()
24    List = {}
```

Line 16 to 24 - Holds all the global vars that the client will need further on to preform all the tasks mentioned above.

Download Files:



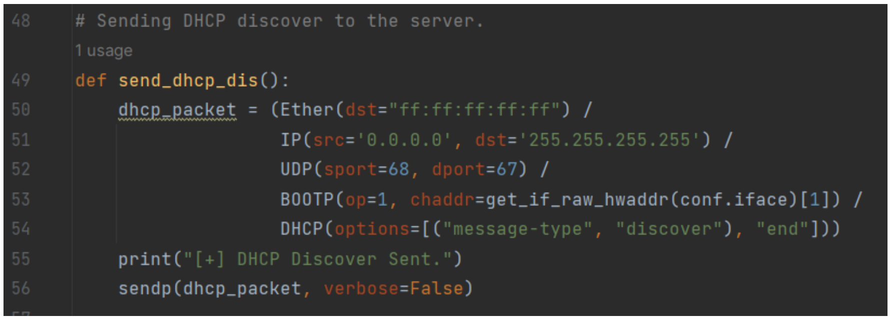
```
27    # Downloading files.
      1 usage
28    def download_file(list):
29        for i in range(0, len(list)):
30            print(f"-{i+1} : {list[i]}")
31
32        while True:
33            num = int(input("The number of the file you would like to download: Press 0 if u want to exit "))
34            if num == 0:
35                print("We are going to exit on the connection with the RUDP Server")
36                print(".................................................")
37                sys.exit(0)
38            if num <= len(list):
39                filename = f"{list[num - 1]}"
40                set_ip_command = f"wget --bind-address={client_ip} http://localhost/{filename}"
41                subprocess.run(set_ip_command, shell=True, check=True)
42                set_ip_command = f"chmod 777 {filename}"
43                subprocess.run(set_ip_command, shell=True, check=True)
44            else:
45                print("Wrong file number.")
```

Line 29 to 30 – Printing the files that's available to download as a list of items using basic for loop.

Line 32 to 37 – Getting the user input that will indicate which files would he like to download.

Line 38 to 45 – Starts by checking if the file number mentioned in the sequence that presented just above and downloading the files right from the HTTP server using the command 'Wget' and followed by the file path.

DHCP Discover Sending:

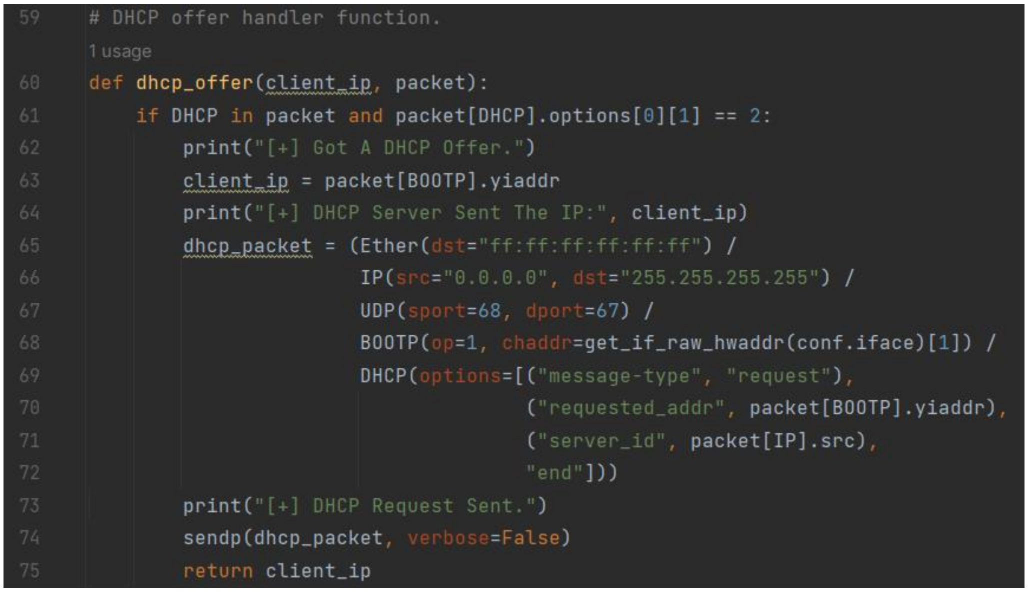


```
48  # Sending DHCP discover to the server.
    1 usage
49  def send_dhcp_dis():
50      dhcp_packet = (Ether(dst="ff:ff:ff:ff:ff") /
51                     IP(src='0.0.0.0', dst='255.255.255.255') /
52                     UDP(sport=68, dport=67) /
53                     BOOTP(op=1, chaddr=get_if_raw_hwaddr(conf.iface)[1]) /
54                     DHCP(options=[("message-type", "discover"), "end"]))
55      print("[+] DHCP Discover Sent.")
56      sendp(dhcp_packet, verbose=False)
```

Line 50 to 55 – Mainly dealing with creating the DHCP discover packet and packing it with the machine information.

Line 55 – Sending the DHCP discover packet made just above and holding the machine info.

DHCP Offer Handler:



```
59  # DHCP offer handler function.
    1 usage
60  def dhcp_offer(client_ip, packet):
61      if DHCP in packet and packet[DHCP].options[0][1] == 2:
62          print("[+] Got A DHCP Offer.")
63          client_ip = packet[BOOTP].yiaddr
64          print("[+] DHCP Server Sent The IP:", client_ip)
65          dhcp_packet = (Ether(dst="ff:ff:ff:ff:ff:ff") /
66                         IP(src="0.0.0.0", dst="255.255.255.255") /
67                         UDP(sport=68, dport=67) /
68                         BOOTP(op=1, chaddr=get_if_raw_hwaddr(conf.iface)[1]) /
69                         DHCP(options=[("message-type", "request"),
70                                       ("requested_addr", packet[BOOTP].yiaddr),
71                                       ("server_id", packet[IP].src),
72                                       "end"]))
73          print("[+] DHCP Request Sent.")
74          sendp(dhcp_packet, verbose=False)
75          return client_ip
```

This function gets two parameters as an argument, client IP and the packet to be handeled.

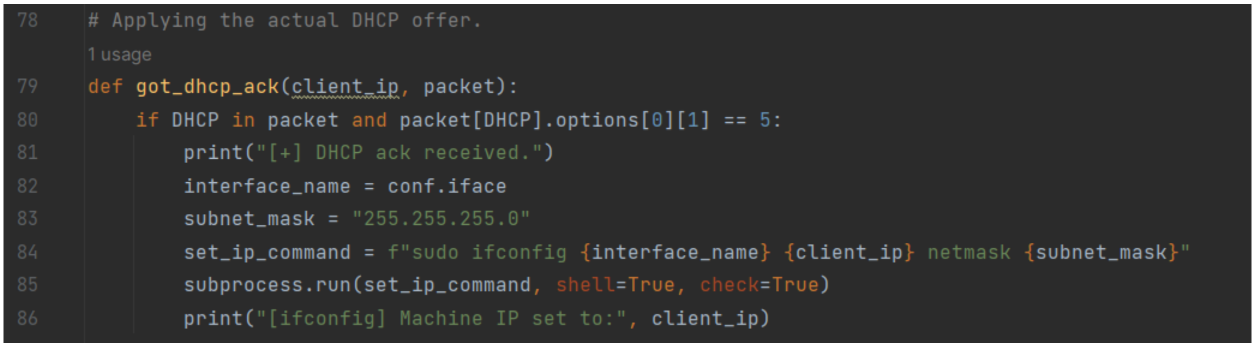
Line 61 – Valiates that the packet that sent in the argument is type 2 means ‘DHCP Offer’ type.

Line 63 – uses the packet to get the client ip from the package.

Line 65 to 72 - Dealing with creating the DHCP packet and packing it with the machine information.

Line 74 to 75 – Sending the packet that was made just above and returning the client_ip to be captured at the main function.

Got DHCP ACK:



```
78    # Applying the actual DHCP offer.
      1 usage
79    def got_dhcp_ack(client_ip, packet):
80        if DHCP in packet and packet[DHCP].options[0][1] == 5:
81            print("[+] DHCP ack received.")
82            interface_name = conf.iface
83            subnet_mask = "255.255.255.0"
84            set_ip_command = f"sudo ifconfig {interface_name} {client_ip} netmask {subnet_mask}"
85            subprocess.run(set_ip_command, shell=True, check=True)
86            print("[ifconfig] Machine IP set to:", client_ip)
```

This function gets two parameters as an argument, client IP and the packet to be handeled.

Line 80 – Valiates that the packet that sent in the argument is type 5 means ‘DHCP ACK’ type.

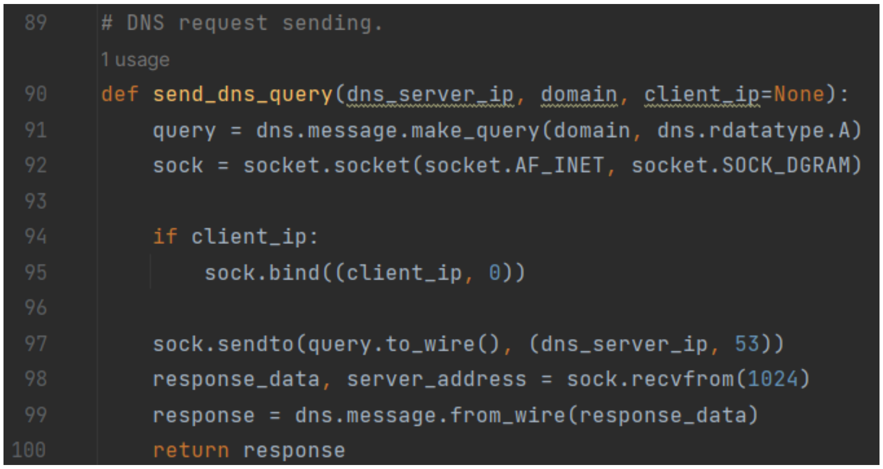
Line 82 - Captures the name of the Network card name.

Line 83 – Defines the Subnet Mask.

Line 84 and 85 – Taking the IP address that was offered from the DHCP and applying it by using the subprocess.run function with the command “sudo ifconfig IP x.x.x.x netmask y.y.y.y”.

Line 86 – Printing to the screen the machine IP to validate the previous applying process.

Send DNS Query:



```
89    # DNS request sending.
      1 usage
90    def send_dns_query(dns_server_ip, domain, client_ip=None):
91        query = dns.message.make_query(domain, dns.rdatatype.A)
92        sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
93
94        if client_ip:
95            sock.bind((client_ip, 0))
96
97        sock.sendto(query.to_wire(), (dns_server_ip, 53))
98        response_data, server_address = sock.recvfrom(1024)
99        response = dns.message.from_wire(response_data)
100       return response
```

This function gets three parameters as an argument, DNS Server IP, Domain and Client IP.

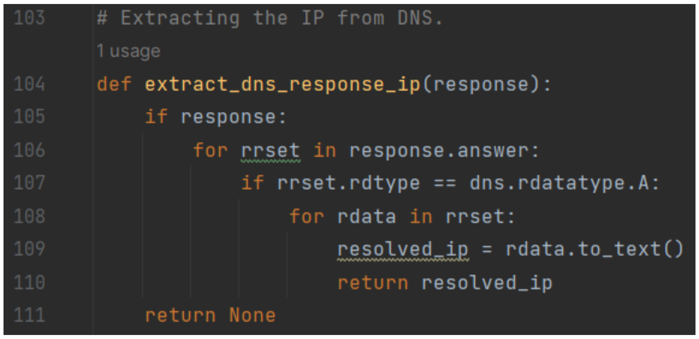
Line 91 – Captureing the dns query using the function dns.massage.make_query, taing two parameters, Dimain and dns.rdatatype.A (stands for dns query for IPv4).

Line 92 – Dns socket declared.

Line 94 to 94 – Cheking for a valid IP address, if the IP is valid, binding the socket.

Line 97 to 99 - These three lines of code send a DNS query message to a DNS server, receive a response message, and sends the response into a DNS message object.

Extract DNS Response IP:



```
103     # Extracting the IP from DNS.
        1 usage
104     def extract_dns_response_ip(response):
105         if response:
106             for rrset in response.answer:
107                 if rrset.rdtype == dns.rdatatype.A:
108                     for rdata in rrset:
109                         resolved_ip = rdata.to_text()
110                         return resolved_ip
111         return None
```

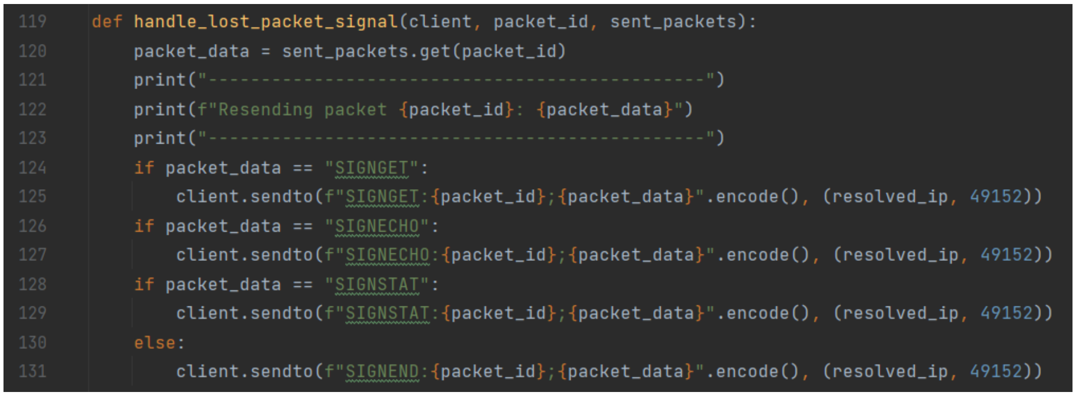
Line 105 to 110 - Extracting the IP address from the response and returns it.

The function searching the response for answer section and looking for A records (IPv4 addresses) and returns the first A record it finds.

If there is no A record to be found the function will return None.

RUDP functions at the client side.

Handle Lost Packet Signal:



```
def handle_lost_packet_signal(client, packet_id, sent_packets):
    packet_data = sent_packets.get(packet_id)
    print("-----------------------------------------------")
    print(f"Resending packet {packet_id}: {packet_data}")
    print("-----------------------------------------------")
    if packet_data == "SIGNGET":
        client.sendto(f"SIGNGET:{packet_id};{packet_data}".encode(), (resolved_ip, 49152))
    if packet_data == "SIGNECHO":
        client.sendto(f"SIGNECHO:{packet_id};{packet_data}".encode(), (resolved_ip, 49152))
    if packet_data == "SIGNSTAT":
        client.sendto(f"SIGNSTAT:{packet_id};{packet_data}".encode(), (resolved_ip, 49152))
    else:
        client.sendto(f"SIGNEND:{packet_id};{packet_data}".encode(), (resolved_ip, 49152))
```

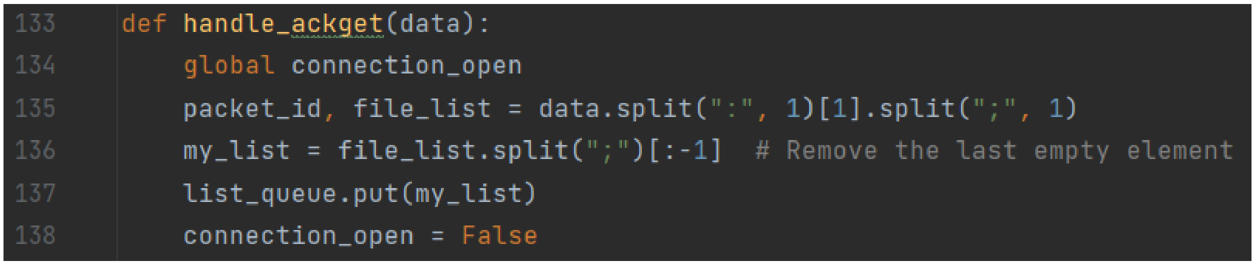
This function use is in case of “SIGNLOST” that the Proxy server sent to the client,

The proxy will send the packet ID of the lost packet to notify the client to re-send that specified packet.

At the client we make an array that holds all the packets by there ID and in case of failure

we will re-send it.

Handle ACK Get:



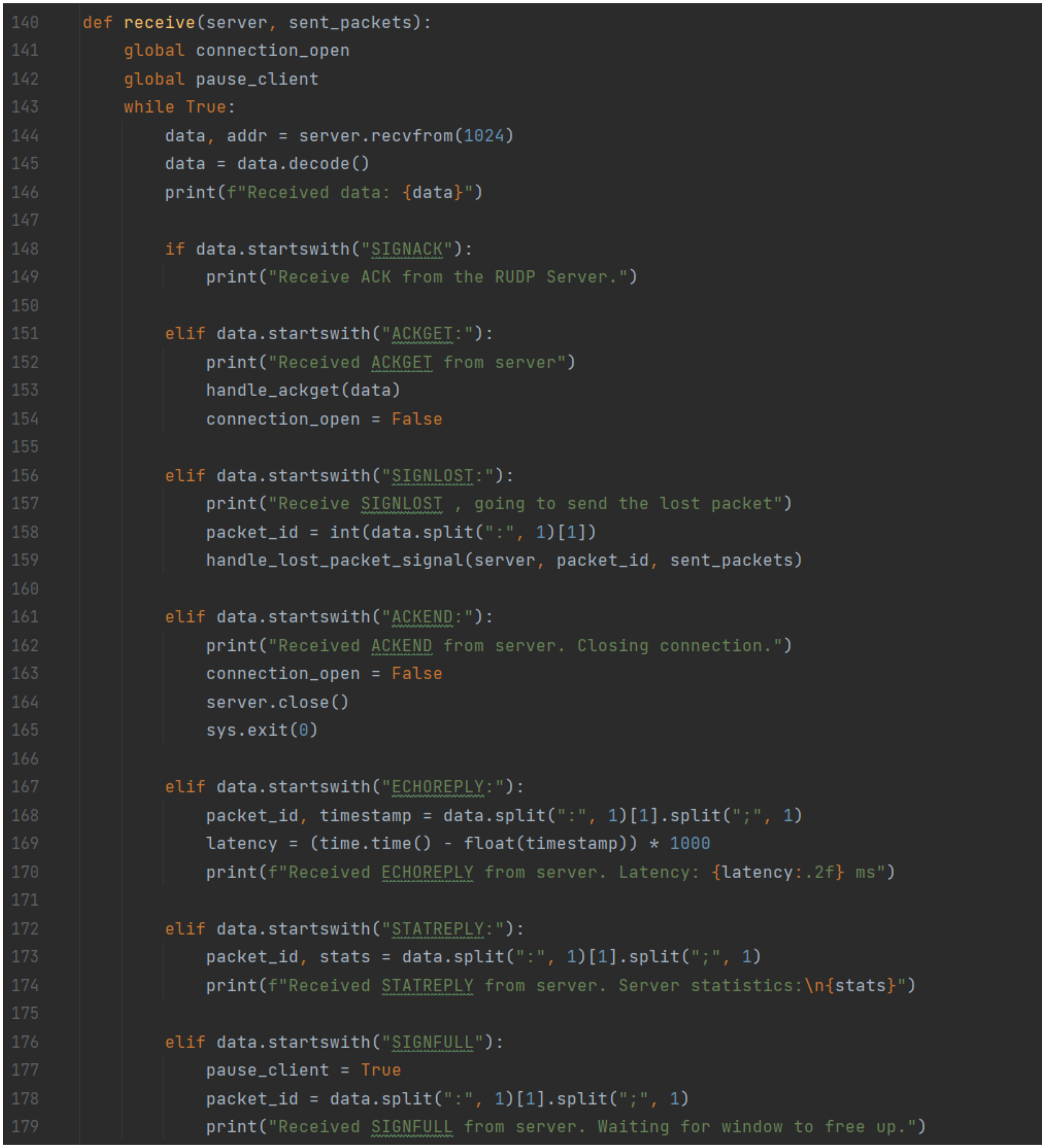
```
def handle_ackget(data):
    global connection_open
    packet_id, file_list = data.split(":", 1)[1].split(";", 1)
    my_list = file_list.split(";")[:-1]  # Remove the last empty element
    list_queue.put(my_list)
    connection_open = False
```

In case that the Proxy sent ACK get this function will come in use. (ACK Get – Proxy answer for the “SIGNGET” that the client sent).

ACK Get also includes the list of downloadable files.

We are adding the list to our Queue.

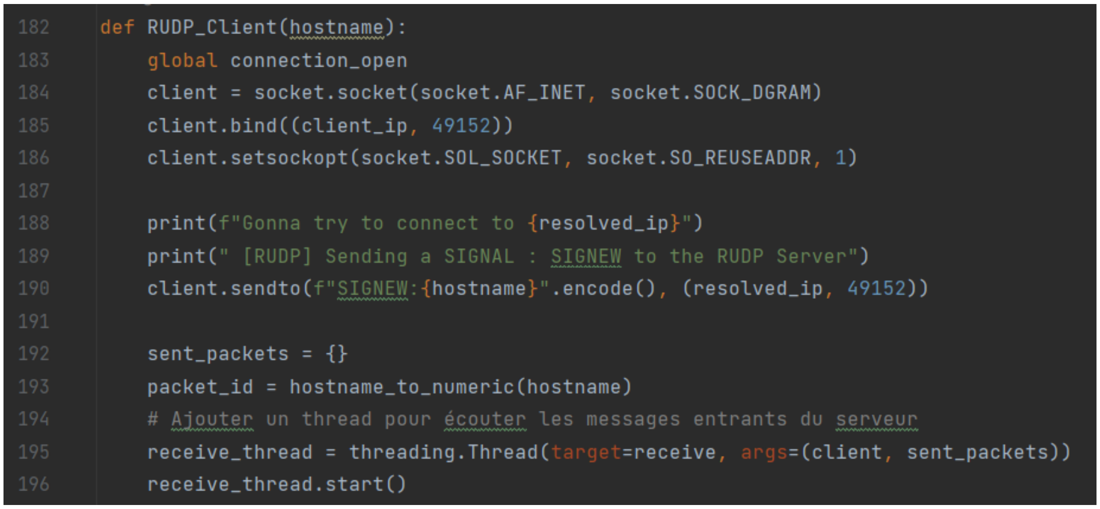
Receive (Thread):



```python
def receive(server, sent_packets):
    global connection_open
    global pause_client
    while True:
        data, addr = server.recvfrom(1024)
        data = data.decode()
        print(f"Received data: {data}")

        if data.startswith("SIGNACK"):
            print("Receive ACK from the RUDP Server.")

        elif data.startswith("ACKGET:"):
            print("Received ACKGET from server")
            handle_ackget(data)
            connection_open = False

        elif data.startswith("SIGNLOST:"):
            print("Receive SIGNLOST , going to send the lost packet")
            packet_id = int(data.split(":", 1)[1])
            handle_lost_packet_signal(server, packet_id, sent_packets)

        elif data.startswith("ACKEND:"):
            print("Received ACKEND from server. Closing connection.")
            connection_open = False
            server.close()
            sys.exit(0)

        elif data.startswith("ECHOREPLY:"):
            packet_id, timestamp = data.split(":", 1)[1].split(";", 1)
            latency = (time.time() - float(timestamp)) * 1000
            print(f"Received ECHOREPLY from server. Latency: {latency:.2f} ms")

        elif data.startswith("STATREPLY:"):
            packet_id, stats = data.split(":", 1)[1].split(";", 1)
            print(f"Received STATREPLY from server. Server statistics:\n{stats}")

        elif data.startswith("SIGNFULL"):
            pause_client = True
            packet_id = data.split(":", 1)[1].split(";", 1)
            print("Received SIGNFULL from server. Waiting for window to free up.")
```

This function that’s running like a “daemon” thread and will run in the background to listen for every response that the Proxy Server made.

Main RUDP Code:

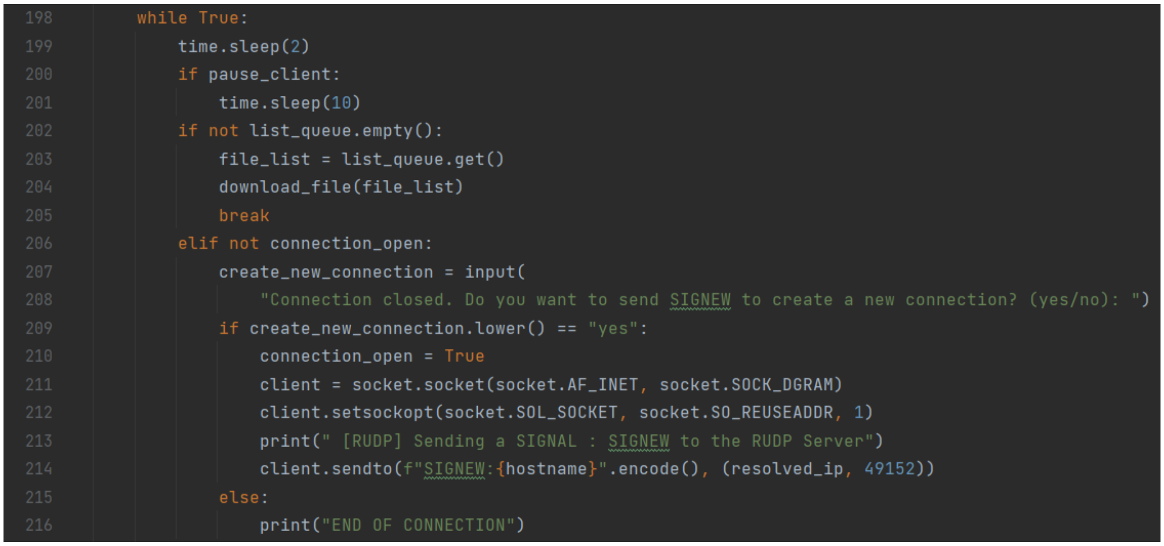


```python
def RUDP_Client(hostname):
    global connection_open
    client = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    client.bind((client_ip, 49152))
    client.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)

    print(f"Gonna try to connect to {resolved_ip}")
    print(" [RUDP] Sending a SIGNAL : SIGNEW to the RUDP Server")
    client.sendto(f"SIGNEW:{hostname}".encode(), (resolved_ip, 49152))

    sent_packets = {}
    packet_id = hostname_to_numeric(hostname)
    # Ajouter un thread pour écouter les messages entrants du serveur
    receive_thread = threading.Thread(target=receive, args=(client, sent_packets))
    receive_thread.start()
```

The first part the setup information for the RUDP Server to be able to connect to the Client.

Plus, we are sending a “SIGNEW” to notify the RUDP that we are about to make a connection.

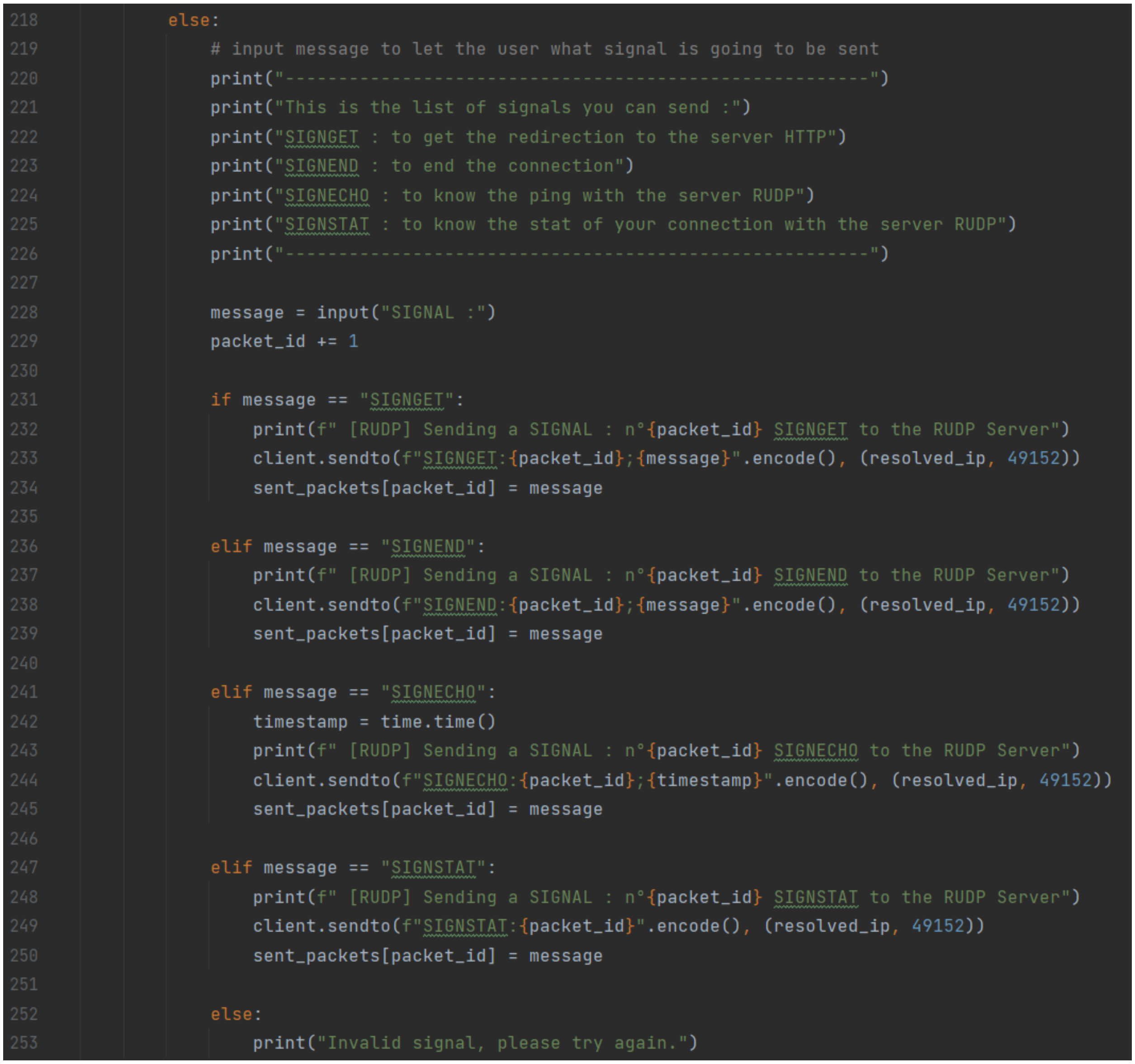
We are starting the thread that will make sure to catch every Proxy response.



```python
    while True:
        time.sleep(2)
        if pause_client:
            time.sleep(10)
        if not list_queue.empty():
            file_list = list_queue.get()
            download_file(file_list)
            break
        elif not connection_open:
            create_new_connection = input(
                "Connection closed. Do you want to send SIGNEW to create a new connection? (yes/no): ")
            if create_new_connection.lower() == "yes":
                connection_open = True
                client = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
                client.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
                print(" [RUDP] Sending a SIGNAL : SIGNEW to the RUDP Server")
                client.sendto(f"SIGNEW:{hostname}".encode(), (resolved_ip, 49152))
            else:
                print("END OF CONNECTION")
```

The while loop here main cause is to make us able to communicate with the RUDP/Proxy Server by showing a prompt up on the screen as type of chat.

In this case the code include if statements to check if the program is in pause mode, if the Queue is not empty and checking the previous connection, in case of disconnection it will establish a new one.



```
            else:
                # input message to let the user what signal is going to be sent
                print("-------------------------------------------------------")
                print("This is the list of signals you can send :")
                print("SIGNGET : to get the redirection to the server HTTP")
                print("SIGNEND : to end the connection")
                print("SIGNECHO : to know the ping with the server RUDP")
                print("SIGNSTAT : to know the stat of your connection with the server RUDP")
                print("-------------------------------------------------------")

                message = input("SIGNAL :")
                packet_id += 1

                if message == "SIGNGET":
                    print(f" [RUDP] Sending a SIGNAL : n°{packet_id} SIGNGET to the RUDP Server")
                    client.sendto(f"SIGNGET:{packet_id};{message}".encode(), (resolved_ip, 49152))
                    sent_packets[packet_id] = message

                elif message == "SIGNEND":
                    print(f" [RUDP] Sending a SIGNAL : n°{packet_id} SIGNEND to the RUDP Server")
                    client.sendto(f"SIGNEND:{packet_id};{message}".encode(), (resolved_ip, 49152))
                    sent_packets[packet_id] = message

                elif message == "SIGNECHO":
                    timestamp = time.time()
                    print(f" [RUDP] Sending a SIGNAL : n°{packet_id} SIGNECHO to the RUDP Server")
                    client.sendto(f"SIGNECHO:{packet_id};{timestamp}".encode(), (resolved_ip, 49152))
                    sent_packets[packet_id] = message

                elif message == "SIGNSTAT":
                    print(f" [RUDP] Sending a SIGNAL : n°{packet_id} SIGNSTAT to the RUDP Server")
                    client.sendto(f"SIGNSTAT:{packet_id}".encode(), (resolved_ip, 49152))
                    sent_packets[packet_id] = message

                else:
                    print("Invalid signal, please try again.")
```

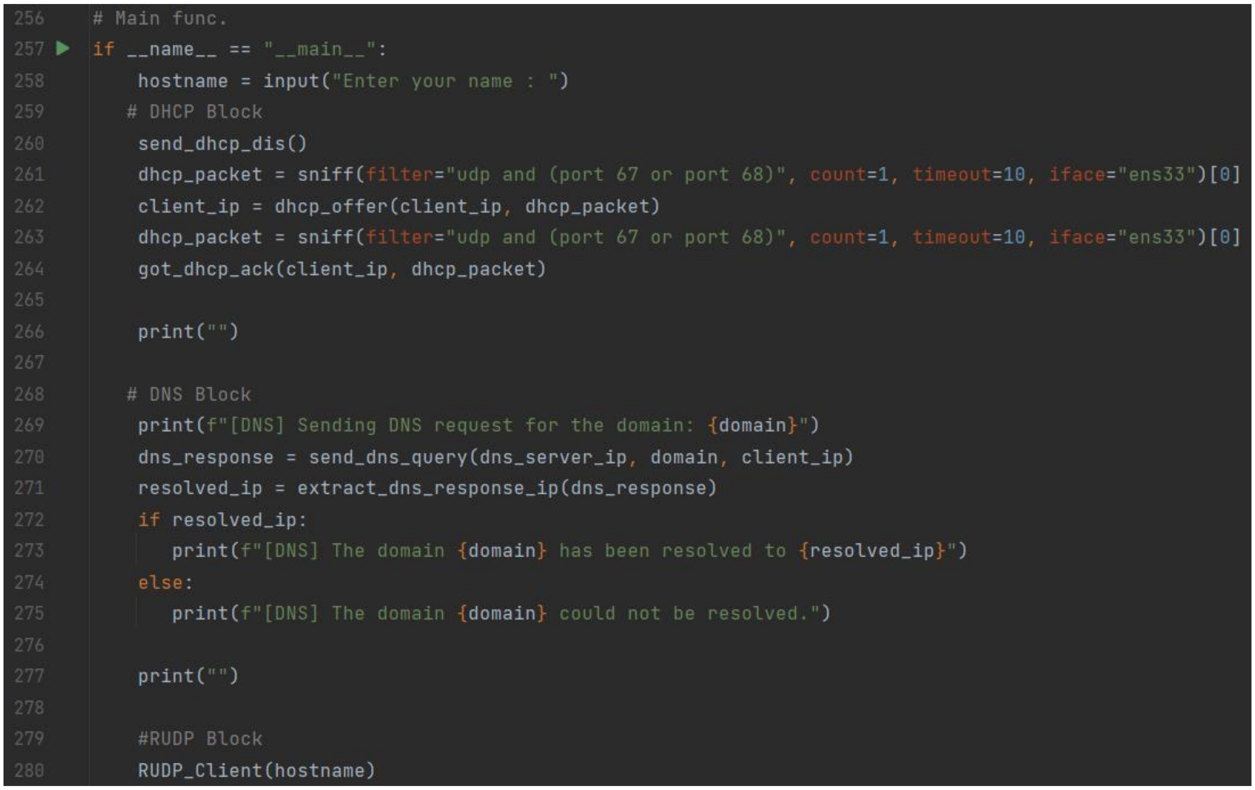
This part main use is “Packet sending”, the right packet to send will be chosen by our pre-defined keywords,

“SIGNGET “, “SIGNEND”, “SIGNECHO”, SIGNSTAT”.

With the explanation for each one of them right above the part of if statements.

Plus, every time that we are sending a packet, we are saving the payload of it in a list by the packet ID to be able to re-send in case of failure.

Main Function:



```
# Main func.
if __name__ == "__main__":
    hostname = input("Enter your name : ")
   # DHCP Block
    send_dhcp_dis()
    dhcp_packet = sniff(filter="udp and (port 67 or port 68)", count=1, timeout=10, iface="ens33")[0]
    client_ip = dhcp_offer(client_ip, dhcp_packet)
    dhcp_packet = sniff(filter="udp and (port 67 or port 68)", count=1, timeout=10, iface="ens33")[0]
    got_dhcp_ack(client_ip, dhcp_packet)

    print("")

   # DNS Block
    print(f"[DNS] Sending DNS request for the domain: {domain}")
    dns_response = send_dns_query(dns_server_ip, domain, client_ip)
    resolved_ip = extract_dns_response_ip(dns_response)
    if resolved_ip:
        print(f"[DNS] The domain {domain} has been resolved to {resolved_ip}")
    else:
        print(f"[DNS] The domain {domain} could not be resolved.")

    print("")

    #RUDP Block
    RUDP_Client(hostname)
```

DHCP Block:

Sending a 'DHCP Discover' to the connected network to be discovered by the DHCP Server,

Then we are sniffing to be able to get the response from the DHCP as 'DHCP Offer' and capture the offered IP as client_ip to apply it further on.

We are sending the 'got_dhcp_offer' packet to notify the DHCP that we got his IP address offer, right after it we are making a sniff once again to make sure that the DHCP sent us back the DHCP ACK and that will be the last step for the IP gathering process.

DNS Block:

We are capturing the DNS packet by using the function send_dns_query and extracting the IP address from the packet we just captured.

Then checking for a valid IP.

RUDP Block:

Firing up the connection between the Proxy/RUDP Server and the Client.